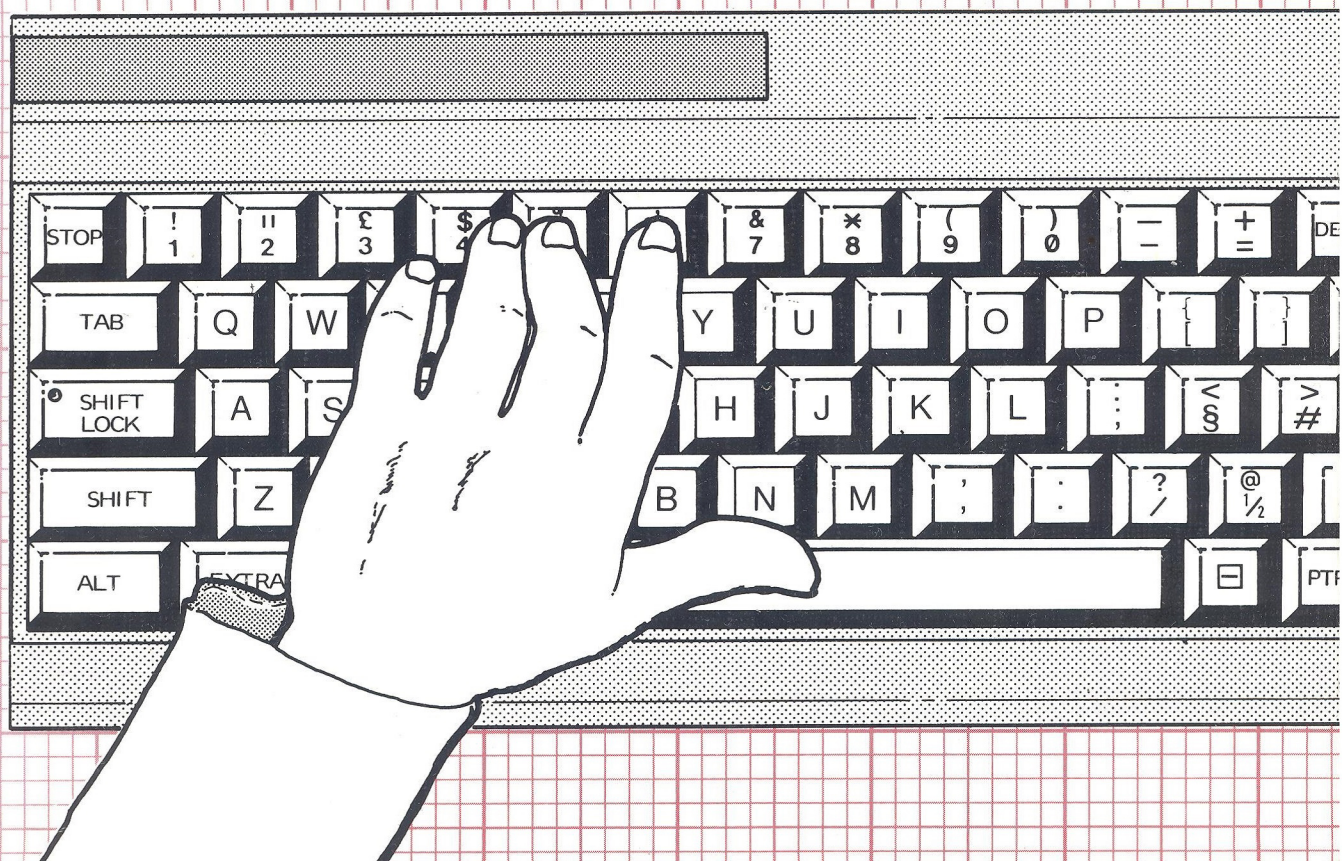


REVISED EDITION
NOW COVERS
8256, 8512 AND 9512

Getting Started with **BASIC and LOGO** on the **Amstrad PCWs**

F.A. WILSON



**Getting Started
with
BASIC and LOGO
on the
Amstrad PCWs**

ALSO OF INTEREST

A PRACTICAL REFERENCE GUIDE TO WORD PROCESSING ON THE AMSTRAD PCW 8256 and PCW 8512

BP187

F. A. Wilson, CGIA, CEng, FIEE, FIERE, FBIM.

Amstrad have now brought word processing within the reach of everyone. These PCW's are capable of manipulating letters and words in practically every conceivable way and what can be achieved with them is only limited by the ingenuity of the user.

With such sophistication it comes as no surprise to find that each model is accompanied by two bulky instruction manuals likely at first to discourage even the most diligent. The information you are likely to need is probably in there somewhere, or is it? Finding it and understanding how to use it may well be another story!

But do not despair – help is here with this indispensable book, written to complement the manuals. It offers much practical advice, simplifies many complicated procedures with specially developed “follow-the-arrow” charts and also includes a comprehensive reference section showing all possible type styles, spacings and margin settings etc.

0 85934 161 5

192 pages

(Large Format)

1986

£5.95

Getting Started with BASIC and LOGO on the Amstrad PCWs

**by
F. A. Wilson**
CGIA, CEng, FIEE, FIERE, FBIM.

**BERNARD BABANI (publishing) LTD
THE GRAMPIANS
SHEPHERDS BUSH ROAD
LONDON W6 7NF
ENGLAND**

PLEASE NOTE

Although every care has been taken with the production of this book to ensure that any projects, designs, modifications and/or programs etc. contained herewith, operate in a correct and safe manner and also that any components specified are normally available in Great Britain, the Publishers do not accept responsibility in any way for the failure, including fault in design, of any project, design, modification or program to work correctly or to cause damage to any other equipment that it may be connected to or used in conjunction with, or in respect of any other damage or injury that may be so caused, nor do the Publishers accept responsibility in any way for the failure to obtain specified components.

Notice is also given that if equipment that is still under warranty is modified in any way or used or connected with home-built equipment then that warranty may be void.

© 1987 & 1988 BERNARD BABANI (publishing) LTD

First Published — March 1987

Revised and Reprinted — February 1988

British Library Cataloguing in Publication Data:

Wilson, F. A.

Getting Started with BASIC and LOGO on the
Amstrad PCWs.

1. Amstrad PC8256 (Computer) — Programming
2. Amstrad PC8512 (Computer) — Programming
3. Amstrad PC9512 (Computer) — Programming
4. BASIC (Computer programming language)
5. LOGO (Computer programming language)

I. Title

005.2'65 QA76.8.A4

ISBN 0 85934 162 3

Preface

Do not fear going forward slowly. Fear only to stand still.

Chinese proverb

It never made economic sense to purchase a machine and then only use half of it so this is a book for all Amstrad PCW users who are experienced in using the machine as a word processor and now wish to explore the computing side. It is also for those who have purchased a PCW with the express intention of using it both for word processing and for computing and for others who know all about BASIC and now want to take up the challenge of LOGO.

Word processing on the 8000-type PCW's is fully covered in the publication BP 187 *A Practical Reference Guide to Word Processing on the Amstrad PCW8256 and PCW8512* hence the two books together cover all the basic facilities of both models.

The manuals contain most of the information necessary for making a success of computing and filing but for the newcomer it is sometimes difficult to see the wood for the trees. Hence a little additional explanation together with plenty of simple exercises is a great help in getting under way. The suggestion is that, having understood the exercises and got them to work as intended, one is in a much better position to work with the manuals for success with the more complicated aspects. From this it is evident that the book is in no way intended as a replacement for the manuals but instead as a useful back up to them. More advanced operations can be undertaken with help from the abundance of magazines and books which exist for this purpose.

The Mallard BASIC has much to offer for it is an advanced form of the language with some special features of its own and with a bias towards business rather than computer games. LOGO is not so well

known to personal computer users but has great promise. It has made a name for itself as a language most suitable for teaching children. This may be so because of its special "turtle" graphic features but the full DR LOGO as used in the PCW's is a wide-ranging powerful language, in many respects an advance on BASIC. It is a pity that the present Guide to LOGO recommended by the manufacturer has been written for different machines so its applicability to the PCW's is incomplete. Much reading between many lines is therefore necessary to get the whole story. There should be no such failing with this present book.

Mallard BASIC, DR LOGO and CP/M PLUS are all trademarks of specialist software firms as indicated in the front of the book. However for convenience, because these titles appear very frequently in the text we use the general terms BASIC, LOGO (and this avoids our minds switching over to the medical profession with which Dr Logo has no relationship whatsoever) and CP/M, recognizing throughout that in most cases the full title really applies.

For convenience the term "8000-type" is used to cover both the PCW8256 and the PCW8512, furthermore the description PCW is deemed to be unnecessary in many cases and is therefore omitted.

Finally a reminder about the title. "Getting Started" means just that. The book is for beginners and those with a modicum of experience. It is not for the expert. Hence with JETSAM in BASIC or recursion and list processing in LOGO we look at the fundamental principles but go no further for it is felt that without ample experience, readers will really be in deep water. These subjects are the province of the more advanced books.

F. A. Wilson – January 1988

ACKNOWLEDGEMENTS

AMSTRAD is a registered trademark of AMSTRAD Consumer Electronics plc.

DR LOGO, CP/M, CP/M Plus are trademarks of Digital Research Inc.

Mallard BASIC and LocoScript are trademarks of Locomotive Software Ltd.

Guide to Logo is published by AMSOFT, a division of AMSTRAD Consumer Electronics plc.

Contents

	Page
Chapter 1. PERSONAL COMPUTERS	1
1.1 Computers and Binary	1
1.2 Memory	2
1.3 Interpreters	3
1.4 The Operating System	3
1.5 The To and Fro Movements of Data	3
1.6 What <i>Do</i> Computers Do?	5
Chapter 2. THE OPERATING SYSTEM	7
2.1 The File Directory	7
2.2 A Worthwhile Shortcut	7
2.3 Preparing for Computing	8
2.4 Copying and Formatting	9
2.5 A Little HELP	9
Chapter 3. COMMON TACTICS	11
3.1 Giving Instructions	11
3.2 Error Messages	11
3.3 Editing	11
3.4 Saving Programs	11
3.5 Character Codes	11
3.6 Variables	12
3.7 Calculations	12
3.7.1 Operator Precedence	12
3.7.2 Random Numbers	12
3.8 And Now For Programming	12
Chapter 4. BASIC BASIC	15
4.1 Direct and Program Modes	15
4.1.1 Line Numbering	15
4.1.2 Editing	15
4.2 Text Layout	15
4.2.1 Screen Width	16
4.2.2 Tabulating	16
4.2.3 Punctuation Marks	16
4.2.4 Formats	16
4.3 Putting It All Together	17
4.3.1 Tracing Faults	18
4.3.2 Printing Out	18
4.3.3 Saving the Work	18
4.3.4 Program Review	19
4.4 Variables	19
4.4.1 Naming Numeric Variables	20
4.5 Data Input	20
4.6 Strings	21
4.6.1 Joining	23
4.6.2 Length	23
4.6.3 Extraction	23
4.6.4 Conversion To and From Numeric	23
4.6.5 Character Repetition	24
4.6.6 Searching Within	24
4.6.7 Case Conversion	24
4.7 Control Characters	24
Chapter 5. BASIC ON THE MOVE	27
5.1 Leaping	27
5.2 Looping	27
5.3 Loop Control	27
5.3.1 FOR / NEXT	27
5.3.2 WHILE / WEND	28
5.3.3 Loops Within Loops	28
5.3.4 Pauses	28

	Page
5.4	Conditionals 29
5.4.1	IF / THEN 29
5.5	Subroutines 29
5.6	Arrays 30
5.7	Program Design 31
5.7.1	Going Around 32
5.7.2	A Flowchart 34
Chapter 6.	BASIC – FILING 37
6.1	Sequential Access 37
6.1.1	Writing To a File 37
6.1.2	Reading From a File 38
6.1.3	Searching Within 38
6.1.4	Alterations 39
6.2	Random Access 40
6.2.1	A Trial Run 40
6.2.2	A Simplified File 41
6.3	JETSAM 42
6.3.1	Keyed Files 42
6.3.2	Setting Up a Keyed File 42
6.3.3	Writing To the File 44
6.3.4	Reading From the File 44
6.3.5	Gilding the Lily 45
6.3.6	Making Changes 45
Chapter 7.	LOGO 47
7.1	The High Resolution Screen 47
7.2	Turtle Talk 47
7.2.1	Graphics v Text 47
7.2.2	Moving the Turtle Around 48
7.2.3	The Turtle Goes Straight 50
7.2.4	Repeat 50
7.3	Printing 52
7.3.1	Enter the Space 52
Chapter 8.	LOGO ON THE MOVE 53
8.1	Procedures 53
8.1.1	Editing 53
8.1.2	Polygons 53
8.1.3	Procedures Calling Procedures 55
8.1.4	Procedures Calling Themselves 55
8.1.5	Calculations 55
8.2	Variables 55
8.2.1	Our Own Primitives 56
8.3	Lists 57
8.4	The Basics, LOGO style 59
8.4.1	The Inevitable IF 59
8.4.2	Looping 59
8.4.3	Pauses 60
8.4.4	At Random 60
8.5	Recursion Recurs 61
8.6	Moving Between Procedures 63
Chapter 9.	LOGO – DATA BASES 65
9.1	LOGO Properties 65
9.2	Creating Properties 65
9.3	Setting Up a Data Base 65
9.3.1	A Data Base Transposed 66
Appendix	SINGLE BYTE BINARY / DECIMAL CONVERSION 69
Index 70

Chapter 1

PERSONAL COMPUTERS

The personal computer (or microcomputer) is built around two essential components, a microprocessor and the memory. The microprocessor can be considered as the device which does all the work and it does it so quickly as to be beyond human comprehension. It is small in size but mighty in output. We leave it at that for it is an extremely complex device. Memory on the other hand is something we ought to know about, what it is, where it is and how much of it there is. However, first a little about the binary system for this is the innermost *language* of the computer and all self-respecting programmers should be acquainted with it.

1.1 Computers and Binary

In our early days we are taught to do our sums in the denary (of ten) system. We usually use the term *decimal* instead. Until computers came along few deviated from this. Computers however do not like the denary system because electrical signals representing any one of 10 numbers can easily be read in error. A computer which makes errors is useless. Hence they use the *binary* system in which, instead of using 10 different symbols (0,1,2,...,8,9), there are only 2. These are labelled by borrowing from the decimal system, 0 and 1. The computer easily recognizes which binary signal is present because a small pulse of electricity represents a 1 and when this is not present it is a 0. It is an on/off arrangement as with an electric lamp, there is no doubt as to which condition it is in, on or off.

One single 0 or 1 is known as a *binary digit* or *bit*. 8 bits are called a *byte*. In the binary system 1024 bytes form a *kilobyte*, not 1000 as might be expected from kilogramme, kilometre etc. Nevertheless 1000 bytes to the kilobyte is often used as a rule of thumb.

The byte is an important unit in computers which generally do their work in blocks of one or more bytes. The Amstrad PCW's are single byte machines as are many current personal computers.

It may not be obvious at first but it is a fact that with less symbols to play with, the binary system needs many more of them to state a number than does the decimal, for example:

decimal	9	27	81
binary	1001	11011	1010001

Unwieldy the binary system may seem at first but in spite of the greater number of symbols, computers can handle huge numbers and leave us humans standing.

A binary number is formed in exactly the same way as a decimal one, but of course the number of symbols to play with is very different, 10 in decimal, only 2 in binary. Taking the decimal number 378 as

an example and with just a reminder that any number raised to a certain power represented by say, n is equal to that number being multiplied by itself $(n - 1)$ times, e.g. $2^3 = 2$ multiplied by itself twice i.e. $2 \times 2 \times 2$:

378 decimal is built up from:

$$(3 \times 100) + (7 \times 10) + (8 \times 1)$$

$$\text{or } (3 \times 10^2) + (7 \times 10^1) + (8 \times 10^0)$$

remembering that 10 (or anything) to the power of 0 is equal to 1.

378 in binary is built up from:

$$(1 \times 256) + (0 \times 128) + (1 \times 64) + (1 \times 32) + (1 \times 16) \\ + (1 \times 8) + (0 \times 4) + (1 \times 2) + (0 \times 1)$$

$$\text{or } (1 \times 2^8) + (0 \times 2^7) + (1 \times 2^6) + (1 \times 2^5) + (1 \times 2^4) \\ + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0)$$

written as 101111010.

Note how the powers of 2 increase by 1 on each move to the left in the binary number just as the powers of 10 do in decimal.

Checking up but first removing any expression containing 0 as a multiplier (because $0 \times \text{anything} = 0$)

$$378 = (1 \times 256) + (1 \times 64) + (1 \times 32) + (1 \times 16) \\ + (1 \times 8) + (1 \times 2).$$

It is useful to know how to convert from decimal to binary and in reverse although at this stage we need not be too concerned for we will see that the Appendix does this for us.

Decimal to Binary: The above example of converting 378 to binary indicates a practical way of doing the job. Note that at each move to the left in a binary number the value of its place doubles e.g.

decimal 1 is 01 in binary
decimal 2 is 10
decimal 3 is 11
decimal 4 is 100

i.e. decimal 3 is indicated by a 1 in the second column to the left having an actual decimal value of 2. There is 1 left over (the *remainder*) hence this occupies the right hand column (or *least significant*) position. The easiest way to convert decimal to binary is in fact by starting at the other end (the *most significant*), successively dividing by 2 and noting

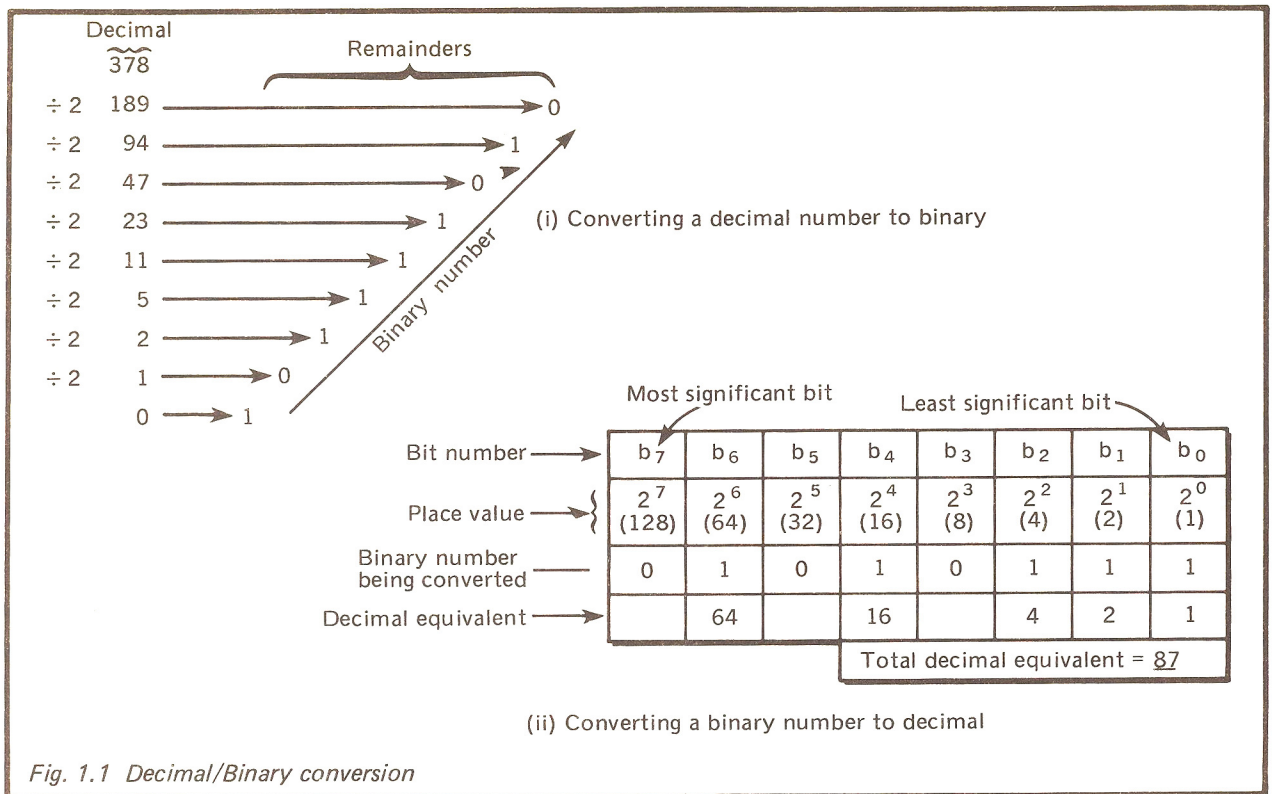


Fig. 1.1 Decimal/Binary conversion

the remainder each time, the binary number then reveals itself. The whole process is illustrated in Fig. 1.1 (i).

Binary to Decimal: We now choose a number lower than 255 which is the maximum which can be contained in one byte (255 in binary = 11111111) so that the computer terminology for one byte held in the memory can also be shown. This is in (ii) of the Figure.

Note that even though the most significant bit in this case happens to be a 0, this is included, the unit is always one byte. As an extreme example, the number 3 held as one byte in the memory would occupy 8 memory cells thus, 00000011. Numbers larger than 255 occupy 2 or more bytes.

Becoming conversant with binary comes with experience. In the early days of learning to program the use of binary does not arise. Nevertheless what we must appreciate is that the microprocessor works in binary *only*, tiny electrical pulses by the million but each and every one meaning something.

Not only are numbers represented by bits (0's and 1's) but so are characters and instructions to the microprocessor. Accordingly every key on the keyboard has its own particular code of 8 bits (there are 256 different combinations of 0's and 1's when the code is 8 bits long). Inside the computer each is sorted out, whether it is an instruction to tell the microprocessor what to do or whether it represents a character to be held for later printing.

The Appendix contains a useful Table from which both way conversions can be made up to decimal 255, i.e. all 8-bit binary numbers.

1.2 Memory

Human memory is the facility we have for storing information and later recalling it. At times we have to admit to forgetfulness, this is usually the recall which is at fault for the information may still be there – somewhere. Computers similarly have memories, information is either already built in or we put it there. The doubtful quality of forgetfulness is not for the computer, recall is always possible. The memory facilities are therefore simply *store* and *recall*. Storing in a computer memory is always in binary, 1's and 0's being held either:

- (i) in tiny electronic circuits which can switch to "on" or "off". These are packed in their thousands into *integrated circuits* (IC's) inside the machine. The PCW's have 256 and 512 kilobyte memories installed within, for example the 256 kilobyte memory contains

$$256 \times 1024 \times 8 = 2,097,152$$

i.e. over 2 million individual switchable electronic cells. Every one of these cells can be used to store a bit and whether the bit is a 1 or a 0 can be determined at any time. Storing in the

memory is known as *writing*, recall as *reading*. The cells are arranged in blocks of 8.

- (ii) the bits can also be stored as miniature magnetic dots and spaces along a magnetic tape or on a disc. The PCW's use the latter method and these are our 3 inch discs. Each disc used with the 8000-type is capable of storing nearly 360 kilobytes. The 9512 goes one better and stores 720 kilobytes. The disc is a form of "portable" memory.

1.3 Interpreters

Although English-speaking people expect all foreigners to learn English, they themselves may not concur. Therefore unless we know their language, for any conversation to be meaningful, a third person must act as an interpreter. Binary is such a foreign language to most of us that the very thought of working directly to a microprocessor in it is most intimidating. Hence the various alternative computer languages which have been developed, in this case BASIC and LOGO. They are both languages which use a vocabulary of English words and mnemonics which the user understands and thereby is able to instruct the computer in what to do. It is evident therefore that an interpreter accepts our English type instructions and data from the keyboard, translates and rearranges as necessary and passes the result to the microprocessor in binary. The interpreter is a highly complex computer program written by experts.

BASIC and LOGO are not the only programming languages available, many others are in use, each having its own special application. To name a few only, FORTRAN is for scientific calculations, COBOL for business, LISP is well suited to artificial intelligence applications and PASCAL was developed for teaching programming.

In personal computers which work only in one language (usually BASIC) the interpreter is held in a special *read-only* memory (ROM). With this type of memory it is not possible to accidentally write into and therefore spoil it. Because the PCW's carry two different languages, one or other of these is read into the computer's internal memory from a disc as required. This sets up the computer for programming in that particular language. Arrangements are made either on the disc case (the write-protection holes) or within the disc so that programs such as these cannot be over-written.

1.4 The Operating System

Helping the microprocessor to do its job of computing is a built-in program known as the *operating system*. This carries out several so-called *house-keeping* jobs which have to be done irrespective of the programming language being used. Such tasks include:

- (i) keeping a constant watch on the keyboard and collecting the data in the form of 8 bits (one byte) as each key is pressed

- (ii) displaying the output from the microprocessor on the screen in English characters
- (iii) arranging for the *saving* of data onto discs or alternatively *loading* from them into the machine
- (iv) arranging for the names of files held on discs to be displayed on the screen i.e. a directory facility
- (v) certain other facilities such as renaming files on discs, erasing them etc.
- (vi) copying and formatting discs and copying files
- (vii) keeping a check on the memory available
- (viii) operating the printer.

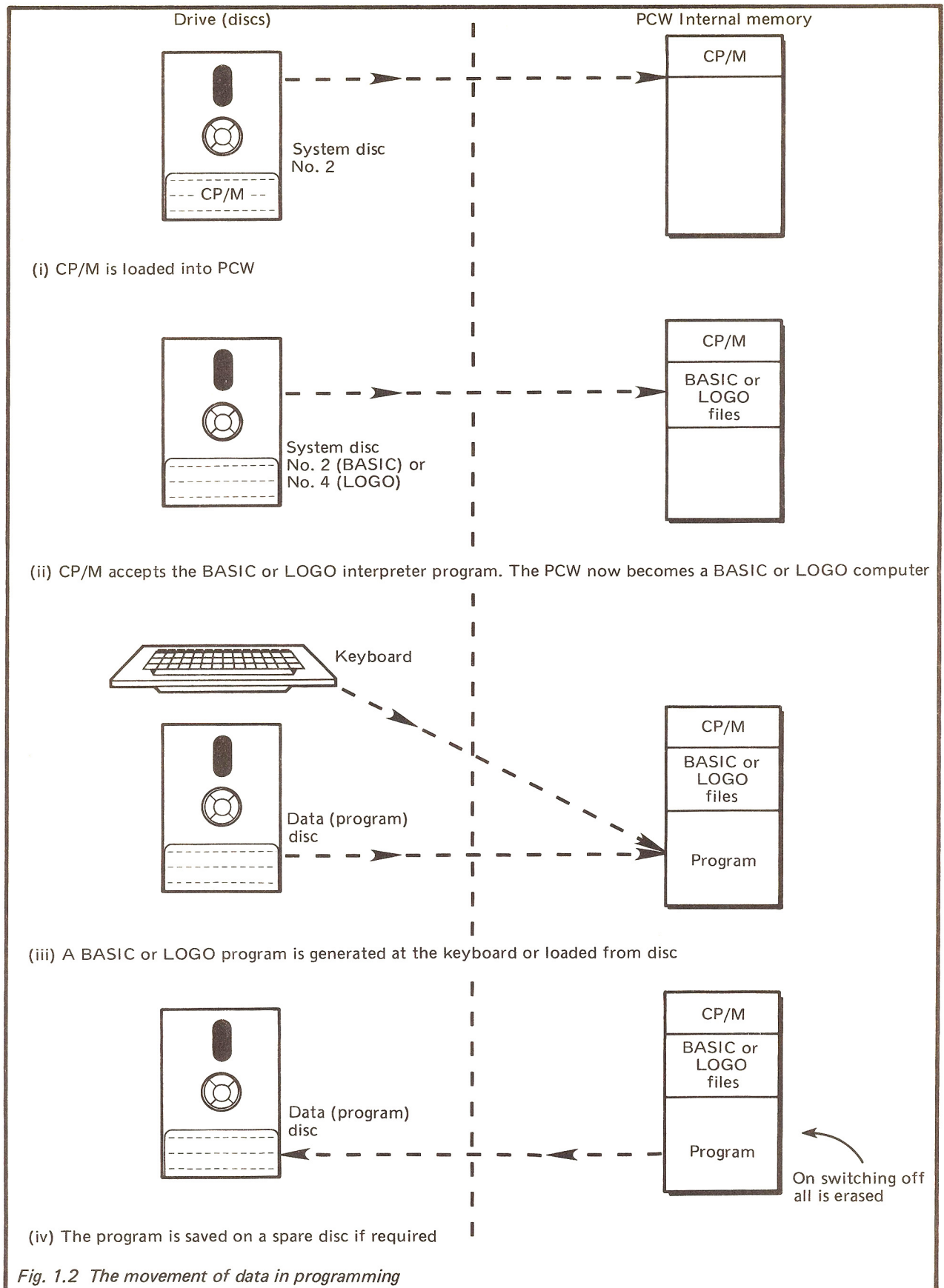
The operating system used is known as CP/M (this particular one has a PLUS added) which is at present one of the standards for 8-bit microcomputers. Accordingly programs written for a particular machine should run on any other 8-bit machine provided that both use the CP/M operating system.

CP/M therefore has to be loaded into a PCW before either BASIC or LOGO programming can commence.

1.5 The To and Fro Movements of Data

Data, that is, the conglomeration of instructions, alphabetical characters and numbers which are the prime ingredients of computing, flows constantly between the internal memory of a PCW and the various discs. Fig. 1.2 shows the flow from switching on to finally saving a new or edited program on disc for safe custody and later re-use for the 8000-type machines:

- (i) – the operating system CP/M is loaded (from System Disc 2) into the PCW's internal memory
- (ii) – the operating system is then called upon to load BASIC from the same disc or LOGO from a second disc (System Disc 4)
- (iii) – now the PCW is set up as a microcomputer and a program can be typed in from the keyboard or alternatively loaded from another disc.
- (iv) – a new program typed in or an edited version of one loaded from a disc [as in (iii)] may need to be saved. It is therefore copied out from the internal memory onto a disc which has sufficient spare capacity.
- (v) – all is lost within the PCW on switching off but any work which is required for later use has been saved on disc and can be re-entered by (i), (ii) and (iii).



The 9512 has everything on two discs, one for word-processing, the other for computing. The latter is labelled "CP/M PLUS" and is used in this case. It contains the Operating System with both BASIC and LOGO. Accordingly with the 9512 no disc-change is required when switching between the two programming languages as Fig. 1.2(ii) indicates is required for the 8000-type.

A program held on a disc can therefore be loaded, changed in any way and re-saved. We will look at all these processes in detail later.

1.6 What Do Computers Do?

Apart from watching the keyboard, writing the stuff on the screen and printing it out, whatever else computers do, it is done with great speed. Computer experts will say that, compared with the big machines, the PCW's are slow, but compared with us, the human race, they are fast, extremely fast. The PCW's can do massive calculations in a fraction of a second which otherwise would take us hours of painstaking work with every prospect of getting the answer wrong. Well, we may not always want to be doing calculations, so, noting that computers obey instructions to the letter with no wandering off and doing their own thing as humans tend to do:

- (i) they can store enormous amounts of information, organize it, shift it about and search for parts we want
- (ii) they can hunt through lists and files to find a particular entry (e.g. FIND in LocoScript)
- (iii) they can be left to make decisions on the course of action to be taken when data is presented to them
- (iv) they can play around with sentences, measure them, cut them up, change them around and even add them together (for how else could LocoScript offer its *Centre* facility?)
- (v) they will sort numbers and letters and put them in the right order
- (vi) they enable us to draw simple pictures and graphs (but not in colour on the PCW's)
- (vii) they will repeat programs of action ad infinitum

- (viii) and they even help us to get our own programs right.

All the above we can try out for ourselves.

The more complicated programs are written for us by experts and these programs are available for a moderate expenditure as *software* (special programs written on discs). Such programs include:

- (i) word processing (such as LocoScript)
- (ii) spelling checkers – the disc contains a library of thousands of words against which our own are checked
- (iii) graphics programs to make the drawing of graphs and bar charts easier.
- (iv) accounts packages – these simplify compilation of invoices, payrolls etc.
- (v) money management – for financial planning and analysis
- (vi) arts packages – these simplify the drawing of charts, diagrams and even pictures
- (vii) tutorial programs – there is an almost endless variety in most subjects and many are especially useful for teaching children
- (viii) and, of course, a multiplicity of games. These normally rely heavily on colour and sound for effects. Those for the PCW's are limited in that colour cannot be used and the only sound available is one tiny squeak.

These lists give us an inkling of the usefulness of computers but there is more. For many people there is the added attraction of writing programs for the sheer joy of it. There is a certain satisfaction in struggling through a program, convinced that it must work and finally getting it to do so. Often this is a greater attraction than using the program itself. It all comes with practice and experience.

Chapter 2

THE OPERATING SYSTEM

Fig. 1.2(i) looks straightforward but apart from setting up the operating system itself, there is work to be done before either the BASIC or LOGO interpreter programs are loaded. On the operating system disc (here labelled under the trademark CP/M PLUS) are many files containing programs and data. Some of these normally would need to be called up one by one for each programming session, a tedious and time consuming activity. Happily CP/M will arrange to do this for us automatically and getting this to happen is one of the aims of the Chapter.

CP/M and the two programming languages work through our giving them *commands* (i.e. instructions as to what to do) by use of English-type *keywords*, each followed by the appropriate data, should any be required.

It is advisable to work with *copies* of the System discs rather than the originals, so if copies are not available, please see Sect. 2.4 first. However, assuming that a copy of System Disc No. 2 (CP/M PLUS etc.) is at hand, switch on then insert the disc into Drive A (on single drive machines there is no other choice). If things do not happen to get going, press any key.

When all has settled down the screen has the heading:

CP/M PLUS Amstrad plc

followed by drive, memory and for the 9512, printer data. Below this on the left is the *prompt*, A>. This is CP/M's invitation to go ahead by entering a command and then pressing [RETURN] (not ENTER as with LocoScript). We are now ready to load the appropriate files.

2.1 The File Directory

The files required are available from the same disc and to see what they are, one of the operating system commands is used. This is DIR for "directory", so type:

dir followed by [RETURN]

We are then confronted by an intimidating list of the files available. Although we may be anxious to get on with actual programming, it is well worth gaining some experience with these first so as to better understand what CP/M is all about. The letter on the left of each line indicates the drive. Each file designation has two parts, a *filename* followed by a *filetype*, many of the latter are the same.

The file with a filename commencing with J also includes the letters CPM and in fact it is the file containing CP/M itself. Most of the other files have the filetype COM and these are all files containing

programs which CP/M runs. Several can be recognized, for example:

DIR this is the file containing the program which has enabled us to display these files

DISCKIT we recall from word processing activities (see also the companion book BP 187) that this is the copying and formatting file (Sect. 2.4).

ERASE and **RENAME** also indicate what their duties are.

With the 9512 there are also BASIC and LOGO in the list. For the 8000-type however there is only BASIC because LOGO is to be found on a separate disc.

When we typed DIR the PCW was not told which drive should have its files listed. Accordingly we got the default drive which is always A. "Default", in the computer world indicates that a predetermined output is given in the absence of instructions to the contrary. To list the files on Drive M, for example would need

dir m: – and don't forget the colon.

2.2 A Worthwhile Shortcut

A search through the list of files will reveal one labelled TYPE.COM (on the screen there is no full stop between filename and filetype but we must always add one). This file contains the program brought into action when the command TYPE is entered. The command is the one which allows us to examine the contents of any file so for experience let us see what is in one of the files, for example the one labelled BASIC.COM, so

type basic.com [RETURN]

brings out the contents and we could perhaps wish we had never started this for there is now a screenful of bewildering hieroglyphics. Clearly at this stage there is no hope of understanding what they mean. Sufficient to say that these characters represent the bytes of instructions used by the microprocessor. However only real experts interfere in this area of computing so we leave well alone. What has been demonstrated however is how CP/M commands are stored in files and used.

8000-type

For users of the 8000-type there is a useful step to be taken which provides a shortcut in the process of setting up CP/M for computing (9512 users please omit the next few paragraphs). From the above it might appear that the command TYPE has

little to offer, but it has, try:

type profile.eng [RETURN]

(PROFILE.ENG is in the top line of the DIR display)

and now 10 of the remaining 23 files are printed out. In fact PROFILE.ENG is not an original file but an assembly of several others. These files are the ones which must be called up each time CP/M is set for computing. To simplify the setting up procedure therefore, arrangements are included for this to be done automatically rather than from the keyboard. It is carried out via the SUBMIT.COM file but this only operates on files which have the filetype SUB. Accordingly to obtain the facility the file PROFILE.ENG must first be changed to PROFILE.SUB. Thereafter a SUBMIT command within the file which loads CP/M, also loads PROFILE.SUB which as we have seen, contains the files of programs required.

To cancel the facility the filetype SUB is changed back to ENG.

Change of file name is obtained via the RENAME.COM file. The command is REN with the new name preceding the old name from which it is separated by an equals sign. Let us carry out the initial setting up procedure step by step:

- (i) either switch on then load a copy of System Disc No. 2 as before OR
if already in action, clear everything by SHIFT +EXTRA+EXIT
- (ii) when the prompt A> arrives, type:

ren profile.sub=profile.eng [RETURN]

- (iii) on the second prompt A>, type:

submit profile [RETURN]

The filetype SUB can be used but is unnecessary because SUBMIT only works with files with a SUB filetype, hence PROFILE.ENG would not be picked up but PROFILE.SUB will.

The required files are copied into Drive M and on the screen they appear one by one and will always do so on setting up for programming (unless PROFILE.SUB is renamed as PROFILE.ENG).

9512

No such juggling is required for the 9512. DIR shows that PROFILE.SUB is already one of the files and it will be found that:

submit profile [RETURN]

produces the same list of files as we get when CP/M is first switched on. When the prompt A> first

arrives therefore, the operating system is already set up for computing.

2.3 Preparing for Computing

To see this fully in action, remove the disc and switch off.

Then:

Switch on, insert the CP/M disc and wait patiently for the final prompt A>. All the 10 files will be listed on the screen.

The PCW is now ready to be told which programming language is to be used.

For BASIC, the file BASIC.COM is on the same disc, so simply type:

basic [RETURN]

and the BASIC screen arrives with the heading:

Mallard – 80 BASIC with Jetsam etc. and showing how many free bytes of memory are available. The BASIC prompt is Ok.

To return from BASIC to CP/M, type:

system [RETURN]

and we go back to the CP/M prompt, A>.

Setting up for LOGO differs for the two types:

9512 – type:

logo [RETURN]

to display the Welcome message and eventually the LOGO prompt, a single question mark, on an otherwise clear screen.

8000's – change the disc for System Disc No. 4 (DR LOGO and HELP) and then type:

submit logo [RETURN]

whereupon the LOGO.COM file is loaded from the disc and the "Welcome to Amstrad LOGO" screen appears followed by the LOGO screen itself with the ? prompt.

With either type of machine, to return from LOGO to CP/M the command is:

bye [RETURN]

again giving the prompt A>.

Just for experience with the 8000-type, while System Disc No. 4 is still in the Drive, type:

dir [RETURN]

and the files contained on the Disc are displayed.

This has completed stages (i) and (ii) of Fig. 1.2 and as the Figure proclaims, the PCW now becomes a BASIC or LOGO computer.

2.4 Copying and Formatting

Mentioned at the beginning of this Chapter is the fact that *copies* of the System discs should be used in preference to the originals just in case one of the latter gets trodden on or even develops internal troubles. In addition, those of our own discs which have become precious should also be copied. The loss of many hours of computing work through a simple accident is too awful to contemplate. CP/M copies discs through the DISCKIT command. Most readers will already have used the word processing facilities and therefore copied the System discs. Hence, because the DISCKIT screen instructions are clear and unambiguous, little is gained by duplicating them here (but see BP 187 for more detail if required). Formatting, that is, preparing a disc for receiving programs, files or other data is equally straightforward. What is more CP/M error messages are of help if we get the discs mixed up.

Switch on, insert System Disc No. 2 (CP/M PLUS) and when the prompt A> arrives, type:

disckit [RETURN]

whereupon the first instructions appear on the screen. Follow the instructions to the letter and the likelihood of failure is small. Finally do not forget that formatting erases all data existing on a disc, hence before using DISCKIT to format it may be advisable to use DIR on the disc as in Sect. 2.1 first. By so doing any files remaining on the disc can be checked.

2.5 A Little HELP

Switch on, insert System Disc No. 2 and when the prompt A> appears, change the disc for No. 4 (DR LOGO and HELP). Type:

help [RETURN]

and the commands on which we can get information from the HELP program are displayed. Next type at the HELP prompt:

commands [RETURN]

There arrives a full explanation of what a command is with an invitation to display a *subtopic* "CONVENTIONS". Type:

command conventions [RETURN]

and a list of the special symbols associated with commands is given. Next try:

help [RETURN]

and the help which is given in HELP shows that in fact to see the COMMANDS topic we only need to type the first two letters, i.e.:

co [RETURN]

and for its subtopic CONVENTIONS

.conventions [RETURN]

the full stop replacing the word "command".

Information on the printer is of interest because we may need to print out our programs later, so from the CP/M prompt A> type:

help [RETURN]

followed by:

pr [RETURN]

which explains the printer command. With the display still on the screen, press [PTR] and it is then possible to experiment with the printer settings. This can usefully be followed by:

.buttons [RETURN] (8000-type)

or **.line 1 (etc.) [RETURN] (9512)**

a useful exercise showing just how much information is available. When finished with HELP, [RETURN] takes us back to the CP/M prompt A>.

There is no need to spend a lot of time examining the individual commands listed under HELP especially as many of them at first may be confusing. All we need to know at present is that help is there and how to access it.

Chapter 3

COMMON TACTICS

To avoid duplication, features common to both languages are first discussed in this Chapter. We can then refer back to them as required.

3.1 Giving Instructions

Programming consists entirely of entering a list of instructions and data into the computer memory. Each instruction may be on a separate line or several may be entered in sequence on the same line.

BASIC does its work through *commands* and *functions* which indicate the actions to be taken. They are typed in by the programmer and are known also as *keywords*. Although the two may appear to be similar, they are not necessarily so, a function is in fact a *mode* of action which in a way amplifies the requirements of the command itself. The difference will soon become clear with experience.

Sometimes what is operated on is known as the *argument*, hence in "do this" or "do that", "do" is the command, "this" and "that" are the arguments.

In LOGO things are very different, all instructions are known as *primitives*, i.e. these are the basic instructions, not derived from anything else. Hence both commands and functions in BASIC are simply primitives in LOGO.

3.2 Error Messages

In typing instructions, errors inevitably creep in. Fortunately both languages help us out with our inadequacies by printing *error messages* when things go wrong. Mostly these arise when the input is unacceptable, e.g. the rules have not been obeyed to the letter or a keyword or primitive has been entered incorrectly. Lists of error messages are to be found in the manuals.

3.3 Editing

We are human, very much so, which implies that getting most programs right at first go is virtually impossible. Accordingly we need to be able to make alterations to programs or in other words, to edit. Both languages contain facilities for change and automatic replacement of the original by an amended version. The actual editing processes however are somewhat different, hence we practise later in Sects. 4.1.2 (BASIC) and 8.1.1 (LOGO).

3.4 Saving Programs

For transferring programs etc. from the PCW internal memory out onto disc so that the data is not lost on switching off, [see Fig. 1.2 (iii) and (iv)], the two languages operate similarly. This also applies to loading the program back in. The program is first given a title and in both cases it is copied out onto the disc by the BASIC command or LOGO primitive, SAVE, followed by the title. In the other direction,

loading the program from the disc is effected by LOAD, followed by the title. There will be experience with this when programs have been generated.

3.5 Character Codes

At the end of Sect. 1.1 the fact is mentioned that every key on the keyboard has an 8-bit binary code. This also applies to key *combinations* (i.e. key A is normally lower case but with SHIFT in addition it becomes capital A). Two separate codes per letter are therefore required. Nowadays the most commonly used coding is specified by ASCII, the American Standard Code for Information Interchange. Standardization is necessary so that computer can talk directly to computer without the need for yet another interpreter. The code covers all letters, numbers and many operations, these are usually accepted by microcomputer manufacturers although frequently home grown codes are also required for special facilities.

The complete set of ASCII codes can be found in any computer reference book while the list of codes for the PCW's is to be found in the manuals under CP/M. However we can manage without for whether in BASIC or LOGO the computer tells us the answer. Any string of 8 bits, although in fact representing, say a character, also happens to be a binary number and this is quoted by the computer in preference to the binary digits. Take, for example, the capital letters PCW. The ASCII codes for these three letters are:

```
P=01010000 (80)
C=01000011 (67)
W=01010111 (87)
```

and these binary numbers all have the decimal equivalents as shown. The latter are easily derived from Appendix 1.

To find the decimal equivalent of the binary code for any character therefore (assuming that the PCW is set for computing as in Chapter 2):

BASIC

```
print asc ("P"); asc ("C"); asc ("W") [RETURN]
```

gives the result:

```
80 67 87
```

ASC being the required BASIC command. Don't be concerned about the brackets and inverted commas at this stage, nor with the dollar sign below. In the reverse direction:

```
print chr$ (80); chr$ (67); chr$ (87) [RETURN]
```

brings the reply PCW.

LOGO Remember, to change from BASIC to LOGO type:

system [RETURN]

and on the prompt A>, change to System Disc No. 4, then type:

submit logo [RETURN]

or just:

logo [RETURN]

for the 9512 (with no change of disc).

The ASCII code commands differ slightly from those in BASIC but the principle is the same. To find the decimal number equivalent to the binary which itself is equivalent to a character, type:

ascii "P ascii "C ascii "W [RETURN]

with same result (80, 67, 87)

Again, in the opposite direction, type:

char 80 char 67 char 87 [RETURN]

and the letters PCW return.

Now all this seems to be a waste of time at present for who needs a computer when we can look up the code equivalents in half the time? It is however useful to understand what ASCII stands for because it appears so often in computer literature. The commands are also frequently employed in programming where controls normally entered from a keyboard have to be applied instead by the program, for example, to clear the screen or move the cursor. The appropriate control is brought into use by quoting its ASCII number. We will see this technique in action in Sect. 4.7.

3.6 Variables

In any programming work comes the need for *variables*. A variable can be likened to a box with a label on the outside, the contents of the box may change but the label does not. The computer recognises this device by the label only. Each time the label is mentioned, the computer looks inside the box to see what its contents are at that particular instant and uses these in the program. It is surprising how useful this technique is and how much programming effort is saved by it. Variables are used to a great extent and will be in action in many of the programs which follow.

3.7 Calculations

At school, perhaps without knowing it, we did our calculations using *operators*. The most commonly used ones are + for addition, - for subtraction, × for

multiplication and ÷ for division. These operations and more are available as standard on practically all microcomputers, the only difference being that the operator signs for multiplication and division are * and / respectively. Computers can easily distinguish between, for example, printing * or using it as a sign that multiplication is required. Accordingly, taking two simple numbers, 8 and 5:

8+5 gives the response	13
8-5 gives	3
8*5 gives	40
8/5 gives	1.6

(with BASIC, to see the results, the instruction is preceded by PRINT).

3.7.1 Operator Precedence

In a multi-operation instruction computers observe certain priorities. Multiplication and division are carried out first, then addition and subtraction. Hence $5 * 2 + 8$ is evaluated as 18, from $5 * 2 = 10$, then 8 is added. The alternative is of course, 50. If uncertain, we can add brackets to those parts which should be evaluated as a whole, so:

$(5 * 2) + 8$ gives 18 (the brackets are unnecessary in this case but nothing is lost by adding them).

or $5 * (2 + 8)$ results in 50.

3.7.2 Random Numbers

These are numbers which arise by pure chance as hopefully they do on a roulette wheel. *Random* implies that any number within a certain range has the same chance of being selected as any other. Both languages have built-in random number generators which in fact are rather complicated mathematical functions, in a way running through a list of random numbers in a fixed pattern. Hence any number is related in some minimal way to the previous number and therefore the two numbers are not strictly *at random*. However the random nature of the output is enhanced by the fact that the generator can be made to start from any point in the list, not necessarily from the beginning. The list is very long indeed so by this method the output can be made sufficiently random for most purposes. Random numbers are employed in some of the BASIC and LOGO programs which follow.

3.8 And Now For Programming

Except for the Appendices the remainder of the book is in two separate parts, Chapters 4 – 6 are concerned entirely with BASIC, Chapters 7 – 9 with LOGO. The two parts are independent of one another hence readers who wish to program first in BASIC continue from here, those wishing to start with LOGO please turn to Chapter 7.

Generally it is assumed in all that follows that the PCW has already been set for programming, if not

please see Sect. 2.3.

We continue with the same keyword conventions, i.e. in the text, keywords are printed in capitals, within a program, in lower case as would be typed. Upper case can be used but there is no point in so doing. However, irrespective of whether we type in upper or lower case, BASIC (but not LOGO) records commands and functions in capitals. For this reason

those programs which have been printed directly from a PCW printer output are in upper case.

In addition, in programming, whatever instruction is given, nothing happens until [RETURN] is keyed, hence, except for the occasional reminder, [RETURN] will be omitted. Square brackets indicate a special *key*, the word is not typed letter by letter.

Chapter 4

BASIC BASIC

Fundamental to any computing system is how to talk back to the user. With personal computers communication is usually by the written word on the screen. From this it follows that one of the most important commands is PRINT and because there are so many variations in the way a printed page (or in this case, a printed screen or *console*) can be set out, it is worthwhile to look at the functions associated with the PRINT command first. To do this however, we must understand the screen itself.

4.1 Direct and Program Modes

There are two *modes* of operation. In the *direct* mode each instruction commences with a keyword followed by the appropriate information. When [RETURN] is pressed the job is carried out but then it is all over, nothing has been retained in the memory. On the other hand the *program* mode is brought into use by merely putting a number before the keyword (e.g. 10 PRINT....). In this case the computer commits the instruction and those which follow, to memory to form the program. This can then be run as many times as required as opposed to only once in the direct mode. To run a program we simply enter:

run [RETURN]

4.1.1 Line Numbering

The computer works through a program strictly in line number sequence. A commonly used practice is to start off by numbering consecutive lines 10, 20, 30 etc. Lines can be added or deleted at will so such numbering gives the opportunity of inserting new lines as required e.g. a line omitted in error between 20 and 30 can be entered as 25. There are two commands which assist in tidy line numbering, AUTO prints the line numbers automatically using the above system of starting at 10 and advancing in tens or there can be any other arrangement if preferred. RENUM is another command which takes an untidy set of line numbers and changes them again to the tens system or any other as required.

4.1.2 Editing

This essential facility enables us to take a line in either direct or program mode and change it using the standard cursor movement and delete keys. In the direct mode the line is presented by [ALT]+A after the [RETURN] for the line has been pressed, for example, type:

print "this is experiment" [RETURN]

and we get what was asked for, but "an" has been left out, so enter:

[ALT]+a

and the line reappears for editing. The word "an" is inserted as in word processing with LocoScript (i.e. by moving the cursor to the e of experiment, then typing **an** followed by a space). [RETURN] then prints the corrected version.

In the program mode the command EDIT is used followed by the line number. When [RETURN] is pressed that particular line is presented and it can be changed as desired. On [RETURN] again the line is entered into the program, replacing the original one. To check that this has happened the command LIST may be used. On its own, all the program is listed, if the command is followed by a range of line numbers, then only these are displayed. Note from LIST how BASIC changes our commands and functions to capitals.

To get rid of a line or lines, use DELETE followed by a single line number or range of numbers, e.g. DELETE 50, DELETE 40-60.

4.2 Text Layout

Each character occupies a certain area on the screen. The screen caters for a maximum of 90 characters per line and 32 lines. For readers who wish to test the screen width for themselves, type the following – and do not forget the quotation marks (also called inverted commas or double quotes):

print "This is an experiment which enables us to see how many characters per line the computer prints out. The 90th character is the r in prints"

On pressing [RETURN] the text is spread across the screen, the top line ending with pr and continuing on the line below with ints. Not very elegant but from this we learn:

- (i) that quotation marks preceded and followed the text to be printed but themselves were not printed. The text is known as a *string* (i.e. words strung together) and whatever is put in a string is printed out exactly, provided that the string is enclosed within quotation marks. If the latter are omitted we are guilty of a *syntax* error and nothing is printed out.
- (ii) the computer is being used in the direct mode
- (iii) ending a line in the middle of a word is hardly good practice. This is easily overcome by breaking up the text. The PRINT command automatically inserts a "carriage return" (i.e. the printing drops to the line below) when the instruction is completed so (using a shorter length of text for convenience), try:

```
print "this is an experiment" : print "to find out"
```

which results in:

```
this is an experiment
to find out
```

showing that text lines can be separated as required. Note that for one command to follow another in the same overall instruction, the second must be preceded by a colon.

4.2.1 Screen Width

We are not stuck with 90 characters per line, there can be less (or even more). Try:

```
width 10 [RETURN] then:
```

```
print "this
is an ex
periment"
```

with the result on [RETURN]:

```
this is an
experimen
t
```

Clearly now we are trying to type on a screen only 10 characters wide. Restore to normal by:

```
width 90 or width 80 etc. as required.
```

4.2.2 Tabulating

There are two functions which start the text at a position other than at the left hand edge of the screen. TAB moves forward to a quoted column number e.g.:

```
print tab(45) "A"
```

prints an A in the middle of the line.

SPC prints a quoted number of spaces e.g.:

```
print tab(35) "P" spc(10) "C" spc(10) "W"
```

spreads the letters PCW in the centre of the screen.

4.2.3. Punctuation Marks

We have already seen in Sect. 4.2 how BASIC uses certain punctuation marks operationally, e.g. quotation marks to enclose a text string, colons to separate instructions on the same line. In addition to these a comma indicates a shift to the beginning of the next *zone*. Unless otherwise instructed, the 90 columns are subdivided into zones of 15 columns each, easily seen by:

```
print 1,2,3,4,5,6
```

the numbers are spread across the screen at 15

column intervals. Zones of 15 are the default size, any other, for example, 10 is given by:

```
zone 10 :print 1,2,3,4,5,6,7,8,9
```

and now the numbers are spread at 10 column intervals. ZONE is a command so a colon is necessary to separate the two commands on the same line.

The semicolon has a different effect, it causes an item to be printed directly after the previous one. The previous character may even be on the program line above. Try:

```
print 1,2,3;4;5;6
```

to display 1 2 3 4 5 6

or this can be done as a program:

```
10 print 1,2,3;
20 print 4;5;6
```

followed by run [RETURN]

which gives the same result. In this case the 4 of the second line is printed adjacent to the 3 of the first line, due to the semicolon after the 3. This is checked by removing it:

```
10 print 1,2,3
20 print 4;5;6
```

when the two sets of figures are printed on separate lines.

4.2.4 Formats

An interesting PRINT variation, slightly complicated, but one which we should at least know exists, is USING to *format* data. A format refers to the style or manner of arrangement of data. This is usually numerical and often the function is employed to keep decimal points vertically aligned, e.g.

```
86.25
2036.87      is not as pleasing to the eye as

86.25
2036.87
```

The decision is first made as to the TAB required, if any and also how many numerals are to be catered for before and after the decimal point. In addition there may be the requirement of separating every 3 figures by a comma. In the program the function USING is followed by a string consisting of one special character per numeral. In the case above the USING format is "####.###" so type in:

```
10 print tab(30) using "####.###"; 86.25
20 print tab(30) using "####.###"; 2036.87
30 print tab(29) using "####.###"; 2036.87
```


(note that USING demands a semicolon after the format data)

The output of this program on entering **run [RETURN]** is:

```
86.25
2036.87
2,036.87
```

Lines 10 and 20 display the information neatly as shown above. Line 30 demonstrates how the comma is entered in the program to separate the thousands figure (2) in the larger number. The TAB therefore needs to be one less.

Repeating the USING format for each entry as in this program would of course be a painful process if long columns of figures were involved. Easier ways of doing such a job come from using other BASIC techniques in addition. There are many other variations with USING as shown in the manuals.

4.3 Putting It All Together

We have now studied enough of the fundamentals of BASIC to be able to set up a simple program and display the results aesthetically. In the following program most of what has gone before is used and commented on. In addition a few other commands which may be needed are introduced. The program is necessarily elementary but there can be no success with more complicated programs unless the simple ones are mastered first.

```
10 width 70
20 zone 14
30 print tab(20) "SATELLITE TV STATIONS"
40 print
50 print tab(13) "INTELSAT"; spc(19) "27.5 degrees West"
60 print
70 print "Frequency", "Channel", "Content", "Schedule", "Scrambling"
```

However, first type:

```
auto
```

This prints the first line number (10) with the cursor waiting in anticipation. Type in the rest of the line to get:

```
10 width 70
```

On pressing [RETURN], the line is entered in the memory and AUTO prepares for line 20. Continue to line 70 as shown and when this line is completed, after [RETURN] press [STOP] (or [ALT]+C) to get rid of AUTO otherwise it keeps wanting to give us line

Just for experience let us first decide whether there is sufficient memory spare for what is about to be done. The function required is FRE, so type:

```
print fre(0) (the 0 has to be there but it is not significant)
```

The number returned shows how many bytes are available. It is over 30,000, enough to keep us typing away all day! Jot the number down (probably more than 31,000) before and after the next program to see how many bytes are used.

In our first and in fact in any program, there will probably be mistakes galore – all the better for it gives plenty of practice in trying out parts and modifying them as we go.

Assume that this is the beginning of a longer program which prints the titles and column headings of a document in which the figures in the columns have yet to be calculated and inserted by the rest of the program. Start by typing:

```
new [RETURN]
```

This is a command which clears everything in the memory ready for a new program so it needs to be used with caution, otherwise usable material may be lost. There can only be one program running at a time, it is therefore a wise precaution to type NEW before starting. The command is not needed however if the PCW has just been switched on. The program to be entered first is as follows:

numbers when they are not required. AUTO need not be used at this early stage and in fact until we are skilled in the art, it can be quite a nuisance! If an error is noticed *after* [RETURN] has been pressed, there is no problem. Simply type EDIT followed by one space then the line number (omit the space and BASIC will advise "Syntax error"). Make the correction(s) and re-enter the line by [RETURN] as shown in Sect. 4.1.2. Take it for granted that this is a process which will need to be used very frequently indeed. Next type:

```
run [RETURN]
```

and the program should print out as follows:

SATELLITE TV STATIONS

INTELSAT

27.5 degrees West

Frequency	Channel	Content	Schedule	Scrambling
-----------	---------	---------	----------	------------

On second thoughts we should give the program a title and perhaps a note of the source of the information although this is not to be printed out. This is accomplished through the command REM. The command simply allows comments to be added to the program but they have no effect on it. BASIC prints them in the program but otherwise ignores them. Quotation marks are not required. For example we might next type in:

```
5 rem Satellite TV
25 rem Information from XYZ Broadcasting
  Authority
```

The [RETURN]s add these two lines to the program as shown by LIST.

Now we have spoilt the numbering sequence so:

```
renum [RETURN]
```

tidies up the program with its output to the screen on LIST finally as shown below:

```
10 REM Satellite TV
20 WIDTH 70
30 ZONE 14
40 REM Information from XYZ Broadcasting Authority
50 PRINT TAB(20) "SATELLITE TV STATIONS"
60 PRINT
70 PRINT TAB(13) "INTELSAT"; SPC(19) "27.5 degrees West"
80 PRINT
90 PRINT "Frequency", "Channel", "Content", "Schedule", "Scrambling"
```

Next, as earlier suggested, let us check how much memory remains free. PRINT FRE(0) now shows that the program occupies some 250 bytes. Plenty left!

If the program does not print out as expected, check especially the spaces, quotation marks, commas and colons, getting these wrong or omitting them is where most errors arise. Bring down the line for edit as already shown.

4.3.1 Tracing Faults

This is a simple program and it is probably easy to see where troubles lie. In more complex programs sometimes a little help may be appreciated. BASIC includes a special command for this, which is TRON, short for "turn on tracing". By entering this command before RUN, the appropriate line numbers are printed on the screen before each line. The command is cancelled by TROFF, so:

```
list brings the program to the screen
tron prepares for tracing
run executes the program but now with the
  appropriate line numbers displayed
troff cancels tracing, otherwise future pro-
  grams will be treated. This is a useful facility
```

although with complex programs it is possible to have too much of a good thing!

4.3.2 Printing Out

Next let us get our work onto paper, the program first. LIST displays the program on the screen but LLIST outputs it to the printer. Putting a sheet of paper into the printer (or pressing [PTR]) brings the printer "buttons" to the bottom of the screen. For the 8000-type machines use the → cursor to highlight "Draft quality", press [+] or [-] to change to "High quality", then [EXIT]. The daisy-wheel printer of the 9512 has no facility for "Draft quality" so for this one no action is required. Then:

```
llist [RETURN]
```

prints out the program.

To print the program *output*, the simplest method at this stage is to change the PRINT commands in the program for LPRINT. This shifts the program output from screen to printer. With the program still displayed, edit lines 50 to 90 by adding an L to the beginning of PRINT. Entering RUN completes the job.

4.3.3 Saving the Work

Some space is available on the System No.2 Disc for

saving programs so this can be used until the message "Disc full" arrives. However it is probably better to start off with a clean formatted disc. The program must have a name which must not contain a full stop because this is the standard way of separating filenames from filetypes (Sect. 2.1). Let us choose SAT_TV for the name so preferably change to the new disc and enter:

save "sat_tv"

and the prompt Ok shows that the program is now *copied* out onto the disc. Next clear the program from the memory with NEW and show that this has been done by the fact that LIST brings nothing to the screen. Then with DIR the program is shown to be on the disc (with filetype BAS for BASIC) and to load it back into the computer type:

load "sat_tv" (BAS is not required with the SAVE and LOAD commands).

To load and run at the same time use:

run "sat_tv"

After the Ok prompt, LIST displays the program again. To erase the program from the disc use the command:

kill "sat_tv.bas" and DIR will show that it has gone.

4.3.4. Program Review

We should now be able to move around in BASIC and set up a program with some experience of the PRINT command and all that goes with it. It may now be worthwhile to retrace our steps and look at this first program line by line, finally bringing into focus all the commands and functions used. The program is in the preceding Section, it may also be on the screen.

line 10 gives the program an overall title. We could have used instead the name chosen for saving.

line 20 reduces the screen width from 90 to 70 characters. It is important to remember that when a print-out is finally required, the *printer* line is only 80 characters (not 90 as for the screen). Reducing the width as we have done displays the printing neatly on A4 paper.

line 30 reduces the zone width to 14 (for 5 zones).

line 40 again is merely for information. REM's can be put anywhere.

line 50 shows how to position the title using TAB and that with functions such as TAB, the number is enclosed within brackets. The text to be printed must be within quotation marks.

line 60 says print nothing i.e. leave a blank line.

line 70 introduces the use of the function SPC.

line 80 (see line 60).

line 90 shows how the comma is used to print the following item at the beginning of the next zone. Note that each individual column title is enclosed within quotation marks so that the commas are not. A comma within a PRINT string is printed as a comma, it then has no operational duties.

Commands and functions now used:-

AUTO, DELETE, DIR, EDIT, FRE, KILL, LIST, LLIST, LOAD, LPRINT, NEW, PRINT, REM, RENUM, RUN, SAVE, SPC, TAB, TRON, TROFF, WIDTH, ZONE.

This Section may be described as laying the foundations for programming rather than actually composing the program itself. The work we have done so far may not be the most exciting, but it is essential. In Chapter 5 things really begin to move.

4.4 Variables

Variables are first mentioned in Sect. 3.6. In this current Section we look at the rules governing them but special programs need not be developed because as the earlier Section points out, variables are used in most programs. The main rules governing the label on the "box" are:

- (i) letters, numbers and full stops are acceptable
- (ii) the first character must be a letter
- (iii) string variables have a dollar sign (\$) appended
- (iv) the variable name must not contain a BASIC keyword.

There are two choices in naming variables. The mathematically minded may prefer to use one or two letters only. This is neat and quickly entered. The alternative is to employ full names because these are more descriptive. The following short programs show the use of variables and also brings out the differences in naming. Suppose it is required to calculate the total price of items when VAT is added

```
10 p=2.6
20 v=0.15
30 t=p+(v*p)
40 print t
```

(* is the multiplication sign – see Sect. 3.7).

Alternatively we could have used the command LET when assigning a variable, for example:

```
10 let p=2.6
```


LET is preferred by some for it perhaps acts as a reminder of the introduction of a variable. It is mandatory on some microcomputers but is not essential in this BASIC.

Here the variable labels are p, representing the price without VAT in £'s, v, the VAT (as a decimal fraction) and t the total price. Running the program elicits the answer, 2.99. Now because p and v are variables, we can change them at will, e.g. by editing line 10 to say, p=5.98 (there are much easier ways of doing this – see Sect. 4.5). Running the program again gives 6.877. This is not a practical amount of money so we now have the opportunity to introduce another function, ROUND which simply rounds the value up or down to a given number of decimal places. Change line 30 to:

```
30 t=round(p+(v*p),2)
```

and we get 6.88. Note that the expression p+(v*p) is followed by a comma then the number of decimal places required, all enclosed in brackets because ROUND is a function.

The more descriptive way of naming the variables might for example be:

```
10 price=5.98  
20 vat=0.15  
30 total=round(price+(vat*price),2)  
40 print total
```

We have the choice. With variables therefore, although these may change, the main course of the program is unaffected and therefore can be used over and over again. Here is an example using string variables:

```
10 ft$="The flight to "  
20 ff$="The flight from "  
30 a$=" arrives at "  
40 l$=" leaves at "  
50 h$=" hrs."  
60 print ff$;"PARIS";a$;"16.05";h$  
70 print ft$;"NEW YORK";l$;"10.35";h$
```

resulting on RUN:

```
The flight from PARIS arrives at 16.05 hrs.  
The flight to NEW YORK leaves at 10.35 hrs.
```

HERE ff\$, ft\$, a\$, l\$, and h\$ are all variable names for different strings (which we recall must be enclosed within quotation marks). Lines 60 and 70 use these strings in the right sequence with other data inserted as required to print out as shown. Note the insertion of spaces in the strings, necessary because an item following a semicolon is printed directly after the preceding item. Again, this is not a practical program because no method of entering information without changing the program is included. This follows in Section 4.5.

4.4.1 Naming Numeric Variables

So far, and many BASIC lovers will already have observed this, we have used names for variables which contain integers without appending the percentage character (%). This BASIC allows us to get away with unless under special circumstances which have not yet arisen. Accordingly we avoid the % where possible in these early chapters. However if the contents of a numeric variable are to contain numbers with fractional parts (i.e. figures to the right of a decimal point), then the final character of the variable name becomes the exclamation mark (!). If in this case we get it wrong and use %, BASIC teaches us a lesson by rounding the value to an integer. Summing up:

```
var$="good"  
var =71  
var%=71
```

var!=71.86 are all acceptable but var%=71.86 is recorded as 72.

4.5 Data Input

To avoid changing the program itself when new data comes along as has happened in the previous programs, a special facility is provided by the command INPUT. When this is used a prompt (?) appears on the screen and the computer waits. Alternatively a text string which has been added to the command by the programmer may be printed and this is followed by a question mark. When the required information has been typed in and [RETURN] pressed, the computer continues with the program. Try:

```
10 input "What is your name please"; name$  
20 print "Welcome to BASIC programming, "; name$
```

On RUN the text in line 10 is printed and the computer stops. The user types in his or her name and presses [RETURN]. The computer gets going again and firstly the name typed in is entered into the string variable which here is called name\$. Then at line 20 the welcome message is printed out. The sequence is as follows:

```
run  
What is your name please? Jane *  
Welcome to BASIC programming, Jane
```

(* the user has typed in her name on the prompt and pressed [RETURN]).

For a user unaccustomed to computers to whom [RETURN] means nothing, line 10 might have been written:

```
10 input "I need to know your name. Will you please  
type it in and press the RETURN key"; name$
```

INPUT need not have a text string included in the

prompt, instructions may have preceded the INPUT command, e.g.:

```
10 print "What is your name please?"
20 input name$
30 print "Welcome to BASIC programming, "; name$
```

The above is a common form of INPUT. There are several variations to the command. Having studied this one, the rest should be easier to understand through the manual and experimenting.

READ and DATA are an inseparable pair of commands used when several items of data are included within a program (not entered during the running of the program as with INPUT). Under DATA the items are listed, separated by commas. READ takes each item from the DATA list and assigns it to a variable of the correct type, i.e, string or numeric. Each variable under READ must have corresponding data under DATA and BASIC moves through these in strict sequence, line by line and along a line. Hence if there are 20 separate data items, READ must have 20 variables quoted to receive them and it is essential that each variable is of the right type to suit its matching data item.

This is clarified by the following program, useless except as an illustration of the various points made.

```
10 data Bournemouth,106,Brighton,53,Dover,77
20 read town1$,a,town2$,b,town3$,c
30 miles$="miles from London"
40 print town1$ " is" a miles$
50 print town2$ " is" b miles$
60 print town3$ " is" c miles$
```

line 10 contains the data items. This line can in fact be put anywhere in the program, BASIC will search for it on meeting READ.

line 20 creates one variable per data item, string variables corresponding to names, numeric variables for the numbers. BASIC assigns each of the data items to its corresponding variable.

line 30 creates a completely separate variable to save a bit of typing later.

line 40 – In this line BASIC meets the first pair of variables (town1\$ and a). Their contents have been derived from lines 10 and 20 hence when the program is run, can be printed out. Miles\$ is not from READ / DATA, as mentioned above, it is a separate variable, standing on its own.

lines 50 and 60 repeat the work of line 40 using the 3rd – 6th data items.

The final result is:

```
Bournemouth is 106 miles from London
Brighton is 53 miles from London
Dover is 77 miles from London
```

Much more can be done with READ/DATA but with programming techniques we have yet to meet, especially in that a computer can select a particular data item from a long list (see Sect. 5.7.2).

There is one other command associated with DATA which is RESTORE. This tells the computer which is the next DATA line to be used, but mainly it is used for restoring to the first line.

4.6 Strings

More can be done with strings than one might imagine, hence we look at them in some depth. It is recalled that a string is simply a sequence of characters. but not necessarily of letters only. Provided that the string is enclosed within (double) quotation marks, when PRINT is let loose on it, the string is printed out exactly as entered, minus the quotation marks. Should a string be assigned to a variable, the name of the latter must end in \$. The following simple PRINT type program serves to remind us of the straightforward use of string variables:

```
10 WIDTH LPRINT 65
20 WIDTH 70
30 INPUT "Name"; name$
40 INPUT "Street"; street$
50 INPUT "Number"; number$
60 INPUT "Town"; town$
70 INPUT "Gift"; gift$
80 LPRINT "TO"
90 LPRINT name$
100 LPRINT number$;" "street$
110 LPRINT town$
120 LPRINT
130 LPRINT
140 LPRINT "Dear " name$
150 LPRINT
```

```

160 LPRINT "What a lucky person you are. You have been specially
      selected in "street$" to receive one of our superb gifts."
170 LPRINT "It is a "gift$"."
180 LPRINT "You don't even have to buy our fabulous book on 'Comp
      uter Bugs'"
190 LPRINT "to claim your gift."
200 LPRINT "Hurry, "name$". Post the coupon today,"
210 LPRINT TAB(15) "Yours sincerely,"

```

the result being, in the case of Mrs Atkins:

```

run
Name? Mrs Atkins
Street? Acacia Avenue
Number? 27
Town? Newborough
Gift? teaspoon

```

On pressing [RETURN] after typing in *teaspoon*, the letter is printed out.

```

TO
Mrs Atkins
27 Acacia Avenue
Newborough

```

Dear Mrs Atkins

```

What a lucky person you are. You have been specially selected in
Acacia Avenue to receive one of our superb gifts.
It is a teaspoon.
You don't even have to buy our fabulous book on 'Computer Bugs'
to claim your gift.
Hurry, Mrs Atkins. Post the coupon today,
      Yours sincerely,

```

So that's how it is done! However there is more to learn from the exercise than just this:

- (i) Yes, computers do have bugs. *Bug* is a somewhat indelicate name for a program error. Removing errors is known as *debugging*.
- (ii) line 10 introduces the command which sets the printer line width. We cannot accurately match screen width with printer line width because each line of the program on the screen must accommodate the line number, command etc. In line 20 a screen width of 70 is a reasonable compromise although some may prefer not to change the width at all. Some juggling with the text strings is inevitable to avoid splitting words at the ends of lines (EDIT,LIST and RENUM are

favourites here). A computer can provide this facility (e.g. as in LocoScript), but certainly not in this program.

- (iii) we have said that strings are printed out exactly as entered. This is not strictly true, there is one exception. Double quotation marks, the indication to BASIC that a string is to follow, cannot be employed within the string otherwise BASIC will end the string prematurely. This is shown in line 180 where single quotation marks are all we have left but are sufficient.
- (iv) note the peculiarity in line 100 of two double quotation marks in succession. This is in fact a string, comprising a single space so that the contents of the variables *number\$* and *street\$* are separated.

- (v) again in line 170 the string there consists of a single full stop. Don't be misled into thinking that gift\$ in this line is within quotation marks, it is not (nor are street\$ in line 160 or name\$ in line 200). The dollar sign tells BASIC that it is a string, therefore quotation marks around string variable names are not required.

4.6.1 Joining

Two or more strings can be joined together to make one longer string. For this the plus sign is used and BASIC adds no spaces at the joints, so one is added here at the beginning of the second string:

```
print "PCW"+" 8256"
```

results in PCW 8256

The numbers in a string are treated only as characters and the + sign is not used arithmetically, hence:

```
print "10"+"66"
```

results in 1066, not 76.

In some manuals the process rejoices in the title *concatenation* (linking together).

4.6.2 Length

There are occasions when it is necessary to know how many characters there are in a string. LEN is the function for this (length) and so for:

```
print len("how many characters?")
```

the response is 20, remembering that space is classed as a character.

Alternatively LEN may be used on a string variable:

```
10 number$="how many characters?"
20 print len (number$)
```

for the same response on entering RUN and again illustrating the golden rule that the argument of a function must be enclosed within brackets.

4.6.3 Extraction

Three functions are provided for extracting parts of a string. LEFT\$ removes and displays characters from the left-hand end of a string, RIGHT\$ from the right-hand end, and MID\$ from within the string. The working of these is most easily demonstrated by examples. We put the string into a variable first to save typing effort and use commas after lines 20 and 30 to spread the results (Sect. 4.2.3):

```
10 quote$="Happiness is no laughing matter"
20 print left$(quote$,9),
30 print right$(quote$,15),
40 print mid$(quote$,14,11)
```

The result is:

```
Happiness      laughing matter      no laughing
```

In lines 20 and 30 the number within the brackets refers to the number of characters to be extracted from extreme left or right. In line 40 the two numbers state where the extraction is to commence and how many characters are involved (not the character number at which the extraction finishes).

4.6.4 Conversion To and From Numeric

It is not possible to operate directly on a number within a string because it exists merely as a set of characters denoting some value. It must first be given back its self-esteem as an arithmetical number before any manipulation can take place. In other words, it must be pulled out from the quotation marks. Suppose there are two text strings (but containing numerals only), "15" and "22" and we wish to add the two numbers together. From Sect. 4.6.1, trying to add these two strings with a + sign results in 1522. What is done therefore is to use the function VAL (value) which changes the string representation of a number to the number itself. Hence VAL("15") becomes the number 15 and:

```
print val("15")+val("22")
```

gives the answer required, 37.

Alternatively the requirement may arise to convert a number into a text string. The function is STR\$ and for example:

```
10 time$=str$(1600)
20 print time$
```

results in the number 1600 preceded by one space.

So far it has been simpler than this e.g.:

```
10 time$="1600"
20 print time$
```

but it will be found that VAL and STR\$ have particular uses during a program. Incidentally, clock times sometimes cause trouble, for if in the above the time is quoted as 16.30, then STR\$ prints out 16.3. BASIC has kindly omitted what it sees as a superfluous nought. There are, of course, ways of getting over this.

As an example of using the two functions together, imagine a string variable named year\$ containing the number 365 and this has to be changed in the program to 366. Line 20 in the following program shows how it is done:

```
10 year$="365"
20 year$=str$(val(year$)+1)
30 print year$
```


Note that the argument of VAL is in brackets and that the VAL statement with the + 1 added becomes the argument of STR\$, hence is also enclosed in brackets. Watch the brackets, they can easily get out of hand and BASIC is especially fussy about them.

```
10 boys$="Tom Dick Harry"  
20 input "What name please"; name$  
30 print instr(1,boys$,name$)
```

and the result is a solitary beep from behind the screen. ASCII 7 is shown as the control code for “bell”, our bell is this high-pitched squeak which in fact is the only sound we can extract from a PCW.

More important is ASCII 27 which is a control code used to initiate an *escape sequence* (ESC). In machine code the escape sequences are instructions of more than one byte, but the first is always ESC (ASCII 27). The remainder of each machine code instruction is made up by various additions to ESC. As an example, ESC followed by the code for a capital E is used to clear the screen. The complete instruction uses the process which joins strings (Sect. 4.6.1) so, if the screen happens to be clear, first type some rubbish on it, then:

```
print chr$(27)+“E”
```

and on [RETURN] the screen clears.

(for interest, the screen is cleared by the two-byte instruction

```
0 0 0 1 1 0 1 1 0 1 0 0 0 1 0 1
  ← 27 →   ← 69 →
  (ESC)      (E)
```

see Appendix 1 for decimal/binary conversion).

Of the complete range of escape sequences quoted in the manuals, at present our interest lies in:

```
CHR$(27)+“E” for clearing the screen
CHR$(27)+“H” to “home” the cursor
CHR$(27)+“Y”; r;c      to move the
cursor to any position on the screen specified by r
(row) and c (column) – [Sect. 4.2]. The range of r is
```

0–31 and that for c, 0–89 and the pair of values required is entered in the instruction in the forms CHR\$(32+r) and CHR\$(32+c). Try this:

```
10 print chr$(27)+“E”
20 print chr$(27)+“Y”;chr$(32+1);chr$(32+87);
30 print “*”
```

Line 10 clears the screen. line 20 sets the cursor position and line 30 prints an asterisk there. On RUN the asterisk appears near the top right corner of the screen (at row 1 and column 87). We could have added together the numbers in the brackets in line 20 if so desired although there are frequently reasons for not so doing.

This appears to be a complicated way of doing a simple job but already we ourselves are capable of reducing the effort. CHR\$(27), which is the beginning of any escape sequence, occurs frequently so why not put it in a variable with a more easily typed name, e.g. c\$, then:

```
10 c$=chr$(27)
20 print c$+“E”
30 print c$+“Y”;chr$(32+1);chr$(32+87);
40 print “*”
```

will do the job just as well and in longer programs less typing will be required.

Let us not be too concerned with the complexity at this stage however, these escape sequences can be assembled as required and put into small individual programs called subroutines (Sect. 5.5). They can then be called up by a single command at any time. A livelier display from a program embodying these features awaits us in Sect. 5.3.3.

Chapter 5

BASIC ON THE MOVE

The programs examined so far are not only short, but also unadventurous. By this it is meant that on the command RUN, the computer starts at the first line and goes through the rest in sequence, then it is all over. We now discover the more lively form of programming, one in which the computer continually moves to different places in the program, often many times. This is its more usual mode of operation.

5.1 Leaping

Within a program BASIC can be told to *leap* (or *jump* or *branch*) back or forth to any other existing line number. Here is the simplest of examples, firstly using the function CHR\$ for more expertise with character codes:

```
10 print chr$(187); chr$(188);  
20 goto 10
```

On RUN the screen is filled with alternate characters ○,● (labelled in the manual as “open circle” and “bullet”). These have the character codes 187 and 188. An alternative program is:

```
10 print “○●”;  
20 goto 10
```

(both characters are available from the full stops key with [ALT])

What happens is perhaps obvious. Line 10 prints one pair of characters then line 20 uses the command GOTO followed by the appropriate line number to tell the computer to do it again and then again and again. The semicolon at the end of line 10 ensures that each pair of characters is printed adjacent to the previous pair. Just to refresh the memory about the use of punctuation marks (Sect. 4.2.3), try changing the semicolon for a comma and the 6 zones immediately become apparent.

Now, unless we ourselves do something, it is evident that this process will continue, even when the screen is full. Although most of the display appears to be stationary, it is not, more characters are being generated at the bottom while the surplus are disappearing at the top. The program can be stopped by the [STOP] key and if required, it can be started again by entering CONT+[RETURN].

Leaping about all over the program via GOTO and other commands yet to be discovered, provides endless flexibility. But it is obvious that more control is needed.

5.2 Looping

In the next step a variable is used in a way which shows how its contents can be continually changed.

Instead of returning to the previous line, the program reverts to the one before that and forms a *loop*. Try (after entering NEW if necessary):

```
10 n=1  
20 print n;  
30 n=n+2  
40 goto 20
```

The computer soon fills the screen with odd numbers and will valiantly continue to do so *ad not quite infinitum* because it does have limits. [STOP] and CONT can be used again if required. After line 10 there is a loop of three lines which does the work of increasing the contents of the variable n by 2 each time and then printing the contents on the screen. There is still no control of the loop except by [STOP], the GOTO instruction is said to be *unconditional* because no conditions are included as to when to cease processing.

5.3 Loop Control

What is required is a means of halting processing at a predetermined point. This is accomplished by several different methods; by telling BASIC (i) to go through the loop a given number of times, (ii) to cease looping when a certain condition arises, (iii) to cease looping when the contents of a designated variable reach a certain level. These techniques bring into play some very useful sets of instructions.

5.3.1 FOR / NEXT

These are two separate commands which always operate as a pair. No GOTO command is required, the loop functions automatically. The FOR command uses a variable of its own as a counter and that variable must be named. In addition the numerical range over which it operates and the numerical step to be taken each time have to be provided. Thus FOR c=1 TO 50 STEP 1 tells BASIC that a loop counter is to be set up, named c, commencing its work with the contents 1 and adding 1 for each transition of the loop up to a maximum value of 50. If no STEP value is quoted, BASIC assumes a value of 1.

Where the loop ends, the command NEXT is entered. To print out the odd numbers up to say, 99 therefore:

```
10 n=1  
20 for c=1 to 50 step 1  
30 print n;  
40 n=n+2  
50 next
```

line 10 *initializes* the variable n i.e. gives it its first contents. A variable must have something in its box, even if it is a 0.

line 20 sets up the FOR / NEXT loop and tells BASIC how to run the counter

line 50 completes the loop.

The FOR command is flexible. As an example:

```
20 for c=100 to 2 step -2
```

does exactly the same job. However, in this particular case where we are playing around with numbers only, why not simply print out the *counter* values?

```
10 for n=1 to 99 step 2
20 print n;
30 next
```

5.3.2 WHILE / WEND

Again an inseparable pair. WHILE marks the beginning of a special loop which terminates in WEND. The loop continues to operate while the expression which controls it is true. The expression is evaluated each time WEND is reached and when found not to be true, processing moves on. The list of odd numbers up to 99 can again be used, we are doing the same job as before but using a different method of control:

```
10 n=1
20 while n<>101
30 print n;
40 n=n+2
50 wend
```

(<> stands for “is not equal to” – see Sect. 5.4.1)

The WHILE loop encompasses lines 20 to 50. It operates while n does *not* equal 101. When at line 40 n becomes 101, this is detected at the following WEND and looping ceases. Processing would then normally move on to the next line but in this case no line exists so it also ceases.

From this arises the opportunity to introduce a special facility for detecting a keypress. The function employed is called INKEY\$ which operates most easily with a WHILE loop. INKEY\$ watches the keyboard to detect when a key is pressed.

The technique is to set up a loop which operates continually during the time that INKEY\$ is receiving nothing. i.e. its output is a *null string*. However looping ceases when INKEY\$ receives a character. A null string is shown in the program as “”.

```
10 print “I am the keyboard. Don’t touch me!”
20 while inkey$=“”
30 wend
40 print “Ouch! I knew you would!”
```

Immediately any character key is pressed, INKEY\$ is no longer equal to a null string (e.g. if k is pressed,

inkey\$=“k”) so the WHILE / WEND loop condition becomes untrue. Looping therefore ceases and the processing moves on to line 40. Lines 20 and 30 would normally be combined:

```
20 while inkey$=“” :wend
```

(not forgetting the colon to separate the two instructions on one line).

5.3.3 Loops Within Loops

Loops can be operated within other loops, the process is known as *nesting*. A program which not only demonstrates this but also amplifies Sect. 4.7 on control characters follows:

```
10 print chr$(27)+“E”
20 for r=0 to 29
30 for c=0 to 89
40 print chr$(27)+“Y”;chr$(32+r);chr$(32+c);
50 print “*”
60 next c
70 next r
```

This prints 30 rows of 90 asterisks, each asterisk being placed through line 40 moving the cursor to a new pair of row and column coordinates. Lines 20 to 70 are in fact a rather complicated way of expressing:

```
20 print “*”;
30 goto 20
```

but they do have full control in that the program ceases when the screen is full.

At the commencement, the counter r in line 20 is set to 0, c is also at 0. Accordingly lines 40 and 50 print one asterisk at position 0,0 (row, column) on the screen. At line 60, NEXT c returns processing to line 30 whereupon c is advanced to 1, followed by printing at 0,1. Again line 60 returns processing to line 30 and this looping (30 to 60 and back) continues until c reaches 89. Not until this final loop is completed can the processing move on and when it does so it meets NEXT r at line 70. This returns to line 20 whereupon r is advanced to 1 and for this the 30–60 loop is traversed another 90 times. Only when r reaches 29 and the c loop within the r loop reaches 89 does the processing cease. Lines 30–60 are therefore a secondary loop within the main loop 20–70.

For a single loop NEXT needs no mention of the variable but for more than one loop, each NEXT must associate itself with its matching FOR by quoting the variable as we have done (NEXT c, NEXT r).

5.3.4 Pauses

A single loop which keeps the computer busy doing nothing is frequently employed to slow things down. It is only necessary to add a self-contained FOR/NEXT

loop (i.e. with no other processing). Try:

```
10 print "*";
20 for t=1 to 1100
30 next
40 goto 10
```

or what is more usual:

```
10 print "*";
20 for t=1 to 1100: next
30 goto 10
```

and now the asterisks, instead of shooting across the screen, have a more leisurely pace. The computer runs the loop counter from 1 to 1100 each time, taking about one second to do so, i.e. a line of 90 characters is printed in around 90 seconds. Hence, knowing that there are some 1100 loop traverses per second, any other delay can be arranged. The timing for any particular machine can be checked by:

```
10 print chr$(7)
20 for t=1 to 10000 :next
30 print chr$(7)
```

On pressing [RETURN] after RUN the first pip is heard immediately. Thereafter the second pip for this computer follows after a time interval of about 9 seconds. The computer takes time in generating the pips but this is negligibly small.

5.4 Conditionals

One thing microprocessing systems are particularly good at is in taking two binary numbers and comparing them for equality. What is more, they can easily decide as to which of the two numbers is the greater. Evidence for this comes from the fact that when a new program line is inserted within an existing program, BASIC immediately puts that line into numerical sequence. Each line number in the existing program is compared with the number to be inserted. At some point in the program, the number when compared with an existing one is found to be greater; compared with the next program number, it is less. Hence this is the point of insertion. Remembering that to a microprocessor characters are represented by single-byte binary numbers (Sect. 1.1), then it follows that characters can also be compared for equality, this is the basis of LocoScript's FIND.

5.4.1 IF / THEN

Comparison of binary numbers is the technique on which one of BASIC's more complicated commands rests. This is the IF command and it allows the computer to make decisions based on criteria entered by the programmer. The full command is in the form:

IF something happens THEN take this action ELSE take some other action.

The ELSE part is optional and is less frequently used. If the something does not happen, THEN has no part to play and programming moves down to the next line. This is a warning not to have the next program instruction on the same line or it will be missed. Frequently the action to be taken after THEN is to branch to another part of the program. Instead of using THEN GOTO..., the THEN may be omitted so the form becomes:

IF something happens GOTO a certain line number ELSE take some other action.

The conditions which apply to the IF command are generally expressed as in mathematics. Those most frequently needed are:

=	is equal to
>	is greater than
<	is less than
<>	is less than or greater than, i.e. is <i>not equal to</i> .

All these symbols are obtained directly from the keyboard.

The IF command also has its uses as a loop control. The uncontrolled loop of Sect. 5.2 which prints out odd numbers can be stopped at say, 99 as follows:

```
10 n=1
20 print n;
30 n=n+2
40 if n=101 then end
50 goto 20
```

The printing instruction is at the beginning of the loop. Therefore if the limit is put as n=99, this particular number gets lost. Alternatively we could use for line 40:

```
40 if n>99 then end
```

The command END terminates a program.

As an example of the use of ELSE:

```
10 input "What is the square of 25"; answer
20 if answer=625 then print "Right first go" else
   print "Sorry you've got it wrong": print: goto 10.
```

The IF command is featured in a practical program in Sect. 5.7.1.

5.5 Subroutines

This is a technique bringing with it much flexibility and avoidance of repetition. A *subroutine* is a miniprogram to which the processor can leap or branch at any time, let the miniprogram do its job, then continue processing from the original point of departure. The command is GOSUB followed by the line number at which the subroutine commences. A second command, RETURN (not the [RETURN])

normally used) signals to the processor that the end of the subroutine has been reached and normal sequence processing is to continue. Subroutines reduce repetition because the same one can be used many times within a program. This in a simple way is demonstrated in the following example in which we try out printing titles in the centre of the page, with spaced underlining of the same length.

```
10 text$="AMSTRAD"
20 gosub 100
30 print
40 text$="Personal Computer Word Processors"
50 gosub 100
60 end
100 shift=int(45-len(text$)/2)
110 print tab(shift) text$
120 print tab(shift) string$(len(text$),95)
130 return
```

the result of this is:

```

      AMSTRAD
      _____
Personal Computer Word Processors
      _____
```

In line 10 the word AMSTRAD is entered into a variable text\$. Line 20 next sends processing to the subroutine commencing at line 100. Now we need to know how far along the line it is necessary to shift the text in text\$ so that it is printed in the middle. If the middle position is 45, then the shift amount must be 45 less half the length of text\$. If LEN(text\$) is odd, dividing by 2 leaves a 0.5 which cannot be used (there is no half a character position). Accordingly the function INT is brought in to convert the number to an integer (whole number) by simply cutting off the fractional part.

Line 110 prints the contents of text\$ at the calculated shift. Starting from the same position, line 120 prints a line. This is Character 95 repeated in a string (Sect. 4.6.5), LEN(text\$) times (Sect. 4.6.2).

Next, line 130 returns processing to the line following that of departure from the main program, i.e. to line 30 which happens to move printing down one line.

At line 40 the contents of text\$ are changed. Notice how entering new contents in a variable completely erases the existing ones. Then follows a jump from line 50 to the subroutine again. The process is repeated with the new text, returning from the subroutine to line 60 where END ceases processing.

Watch out for syntax errors caused by getting brackets wrong. In any instruction there must be an equal number opening and closing.

There can be any number of subroutines attached to a main program, each one identified by its first line number. Subroutines themselves may contain other subroutines.

5.6 Arrays

Strictly arrays are an extension of the technique of variables. A discussion of them is not included in Sect. 4.4 however because at that stage other procedures which are helpful in demonstrating the advantages of arrays have not been examined. This applies especially to FOR / NEXT loops.

There are occasions when many variables are required, all having some common basis. Take for example, the problem of recording monthly statistics. The 12 months have one feature in common, they are all part of a year. Assuming string variables, one way to handle them would be to create 12 separate ones, jan\$, feb\$ etc., an unwieldy process, especially when it comes to printing out. BASIC comes to the rescue with its DIM command by which are entered the *dimensions* or size of an array. An array might therefore be defined as a well-ordered series of things. In the instruction the group of variables is given an overall name, followed by the array dimensions in brackets, e.g.:

```
dim year$(12)
```

This tells BASIC to reserve sufficient space for 12 variables under the general string variable name, year\$. The 12 variables are allocated their contents directly, through READ / DATA or INPUT as the occasion demands. If directly, we would enter:

```
year$(1)="January".....to.....year$(12)="December"
```

and for example, PRINT year\$(3) would result in *March*.

This is the simplest form of array, they can also be set up on the basis of row and column, e.g. an array dimensioned by 3,6 would encompass 3 rows each of 6 variables, a total of 18.

To demonstrate the use of such a *rectangular* array, here is a program which develops the 1–12 multiplication tables so that where row and column meet is the answer, as shown in Fig. 5.1. There are, of course 12 rows and 12 columns:

```
10 dim ans(12,12)
20 for a=1 to 12
30 for b=1 to 12
40 ans(a,b)=a*b
50 next b
60 next a
70 print chr$(27)+"E"
80 for row=1 to 12
90 for column=1 to 12
100 r=2*row
110 c=6*column
120 print chr$(27)+"Y";chr$(32+r);chr$(35+c);
```

	1	2	3	4	5	6	7	8	9	10	11	12
1	1	2	3	4	5	6	7	8	9	10	11	12
2	2	4	6	8	10	12	14	16	18	20	22	24
3	3	6	9	12	15	18	21	24	27	30	33	36
4	4	8	12	16	20	24	28	32	36	40	44	48
5	5	10	15	20	25	30	35	40	45	50	55	60
6	6	12	18	24	30	36	42	48	54	60	66	72
7	7	14	21	28	35	42	49	56	63	70	77	84
8	8	16	24	32	40	48	56	64	72	80	88	96
9	9	18	27	36	45	54	63	72	81	90	99	108
10	10	20	30	40	50	60	70	80	90	100	110	120
11	11	22	33	44	55	66	77	88	99	110	121	132
12	12	24	36	48	60	72	84	96	108	120	132	144

Ok

Drive is A:

Fig. 5.1 A multiplication table array

130 print ans(column,row)
140 next column
150 next row

Line 10 tells the computer to prepare for 144 separate numerical variables under the overall name *ans* (for "answer"). Their individual names are therefore *ans*(1,1), *ans*(1,2).....*ans*(12,11), *ans*(12,12).

Lines 20 to 60 contain two FOR / NEXT loops, one nested (Sect. 5.3.3). The two loops together run through the 144 variables with line 40 producing the contents of each and multiplying them together. As an example, for the 7th operation of the *b* loop within the 5th of the *a* loop, the particular variable under consideration is *ans*(5,7). The two numbers are multiplied together to produce the variable contents, 35. After line 60 the contents of all 144 variables have been entered.

Next it is necessary to print out the variable contents but BASIC has yet to be told how to print them out in the required 12×12 square. This is the second part of the program and it would appear to be as complicated as the first (loops within loops can be very puzzling, especially when they refuse to work!). To print out each number neatly on the screen the control characters are required. (Sect. 4.7). This happens in line 120 for row and column numbers obtained basically from the FOR / NEXT loops com-

mencing at lines 80 and 90. Line 100 spaces the rows two apart and line 110 starts each column 6 spaces to the right of the previous one. Line 70 clears the screen.

Line 130 prints out each variable at the position determined by line 120, according to the current loop values. As an example, when at line 80, row=9 and at line 90, column=4, line 130 prints the contents of *ans*(4,9), i.e. 36.

The unexpected 35 at the end of line 120 (we are more used to 32) simply shifts the whole display 3 positions to the right so that the screen margins are similar.

Printing the tables out would normally be via LPRINT, but we might now try the special printing facility which copies the screen directly onto paper (not available on the 9512). Load paper into the printer, select "High Quality" (if required – Sect.4.3.2), press [EXIT] then [EXTRA] (first) with [PTR].

Just for experience, try TRON (Sect. 4.3.1) before RUN and it is possible to get lost in a proliferation of line numbers as the loops get busy. Escape back to normal with TROFF.

5.7 Program Design

Surely in trying out the foregoing short programs, one inescapable fact has been brought to light – it is

so easy to make mistakes. True but they are also easily rectified through EDIT, LIST, DELETE and RENUM. The last one is a particularly helpful command because it also updates all references to line numbers automatically (GOTO, GOSUB etc.). There is no fear of damaging the PCW unless of course, frustration reaches the point where we throw the keyboard out of the window. But seriously, for most of us, it is not sensible to start at line 10 and expect to plod through line by line to the end, there must be some earlier planning. In this respect a *flowchart* of sorts is useful although not all expert programmers would agree. But certainly programming is a handiwork of personal style.

In this Section we first look at a program which has been designed to give practice in many of the basic techniques already discussed. These are, manipulating strings, subroutines, use of control characters, FOR /NEXT, IF/THEN/ELSE, WHILE/WEND and random numbers. To be honest, the outcome of the program is somewhat ineffectual unless small children are available, but more useful programs can be built up later on these same principles. What is lacking however in this program is a call for READ/DATA, so for this a second, small program follows with an example of a flowchart. This also fulfils the promise made in Sect. 4.5 to develop a program showing how data can be selected.

5.7.1 Going Around

This program while embodying all the tactics men-

tioned above is also a good example of how processing can leap or branch to so many different places. The actual number of moves made by the processor is therefore far greater than is indicated by the number of program lines. From this it is evident that some dedication is required in design and debugging.

The outcome of the program is to test a small child's capability in multiplication of pairs of numbers, say, up to 12. The two numbers are presented randomly so nobody (except BASIC) knows what is coming. We might perhaps first look at the main blocks of processing required:

- (i) ask the child if he or she wants to play
- (ii) ask for the child's name so that the computer can be friendly by using it
- (iii) put up the test on the screen
- (iv) accept the child's answer
- (v) check the answer and make the appropriate remarks
- (vi) return to (iii) with a different pair of numbers.

(if these remarks were abbreviated and written into boxes with arrows pointing to the next box in the sequence, we would have produced a *flowchart* of sorts). Here is the program:

```

10 REM MULTIPLICATION TEST
20 GOSUB 330
30 PRINT "Do you want to see how good you are at multiplication?"
40 PRINT
50 PRINT "If YES, press any key"
60 IF INKEY$="" THEN GOTO 60
70 GOSUB 330
80 INPUT "Type in your name please and then press the RETURN button...", me$
90 REM next 2 lines change first letter to capital
100 first$=LEFT$(me$,1)
110 name$=UPPER$(first$)+RIGHT$(me$, LEN(me$)-1)
120 PRINT
130 GOSUB 330
140 PRINT "Off we go then, "name$". Press any key to start"
150 GOSUB 400
160 GOSUB 330
170 PRINT "multiply ";x;" by ";y
180 PRINT
190 INPUT "Please type in the answer and press RETURN.....", try
200 PRINT
210 ans=x*y
220 REM next line checks answer and decides action
230 IF try=ans THEN GOSUB 350 ELSE GOTO 300
240 PRINT "Good ";name$. Right first go"
250 FOR t=1 TO 2000: NEXT t
260 PRINT
270 PRINT "Press any key for another try"

```



```

280 GOSUB 400
290 GOTO 160
300 PRINT "Oh dear ";name$;" , you've got it wrong. The answer is "ans
310 GOTO 250
320 REM subroutine to clear screen and home cursor
330 PRINT CHR$(27)+"E"+CHR$(27)+"H"
340 RETURN
350 FOR count=1 TO 8
360 PRINT CHR$(7);
370 NEXT
380 RETURN
390 REM subroutine to produce 2 random numbers
400 WHILE INKEY$=""
410 x=INT(RND*12+1)
420 y=INT(RND*12+1)
430 WEND
440 RETURN

```

Going back to the above 6 requirements, (i) is taken care of by firstly clearing the screen by line 20 which uses the subroutine at 330 and 340 (Sects. 4.7 and 5.5), then printing the question on the screen by line 30. Using the idea of pressing a key to continue the program is controlled by line 60, with a slightly different approach from that in Sect. 5.3.2 but equally effective. The processor stays on this line, continually checking INKEY\$ until a key is pressed whereupon there is no longer an empty string. THEN can be omitted from the instruction.

The next requirement (ii), is satisfied by line 80, assuming that a small child can hunt successfully for the correct keys to type in the name. On the further assumption that the typed-in name will not commence with a capital letter, the first letter is changed to a capital one at lines 100 and 110. We saw what can be done with strings in Sect. 4.6 and these two lines take out the first letter, change it to a capital and then add to it the remainder of the name. After clearing the screen again at line 130 and waiting for the signal to continue (line 140), the processor moves on to line 150 and thence to the subroutine at lines 400–440.

This needs a little more explanation about random numbers than is given in Sect. 3.7.2. RND on its own in BASIC returns a random number between 0 and 1 (actually 0 is possible, but 1 is not). The range of numbers required here is from 1 to 12 and they must be integers. INT simply throws away any decimal fraction so multiplying the RND output by 12 therefore produces a range from 0 to 11. Accordingly, adding 1 to an output from INT(RND*12) changes the range to 1 to 12 inclusive. Also in the earlier Section it is indicated that the random number generator should be called upon to produce numbers some way down its long list because starting from the beginning of the list each time produces the same series of numbers. Hence advantage is taken of the fact that the time taken to

press a key after the decision to do so is itself fairly random. Bearing in mind that the generator is moving very quickly through an extremely long list of numbers (very many thousands), a number taken by this method will bear practically no relationship to any used previously. The variables x and y receive random numbers running down the list until a key is pressed. As soon as this happens, INKEY\$ is no longer equal to a null string and x and y take on the random numbers at that moment (see Sect. 5.3.2 for WHILE/WEND).

Requirement (iii) is satisfied on the return to line 160.

For (iv) the answer is put into the numeric variable *try* (line 190) and for (v) the correct answer is calculated at line 210 then checked against the input answer at 230 (Sect. 5.4.1). If the input answer is correct, processing moves to the subroutine at line 350 to cause the PCW to give 8 little squeaks of delight, returned by line 380 to 240 where the result is printed on the screen. If the try is incorrect, line 230 moves processing to 300 for the commiserations to be printed. In both cases processing moves on to line 250 for the next try [requirement (vi)]. A short pause is included at this line (Sect. 5.3.4) and the random number procedure is repeated before returning to line 160.

It all looks very complicated and when we first start there are times when it seems that although the program *should* work, it just will not. TRON and TROFF (Sect. 4.3.1) must not be forgotten but equally STOP is useful. There are two STOP's, the [STOP] key halts a program at the moment of operation of the key and is used for "escaping" from a program, the STOP *command* is inserted within a program so that we can see just what has happened up to that point. As an example, in the current program, if lines 100 and 110 do not seem to play the game, adding, for example:

```

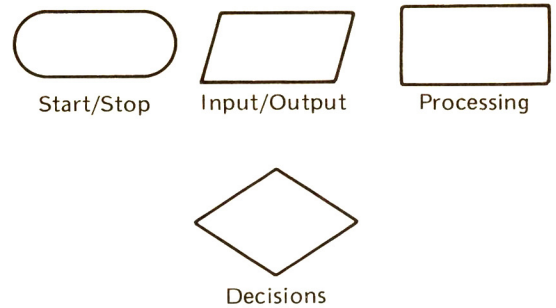
105 print first$
115 print name$
116 stop

```

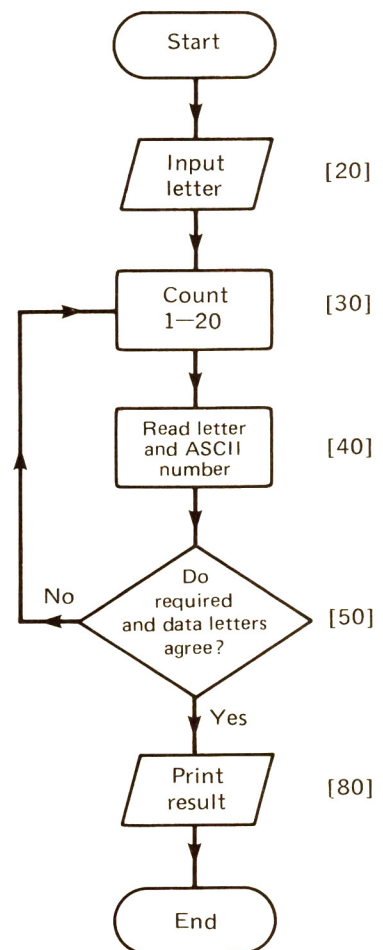
shows each step and helps in seeing whether it is producing the desired result. When sorted out, DELETE 105, then again for 115-116 takes these debugging lines out. To restart a program after STOP of either type, use CONT (for continue). Also with RUN it is not necessary to go through the whole program, simply follow RUN by the line number from which to start.

5.7.2 A Flowchart

Although the output of this second program duplicates a facility already provided on demand by BASIC, it is useful as another exercise involving READ / DATA. It prints out the ASCII number for any character which is entered (but for convenience the character range is limited). This gives an indication as to how the big machines work for say, flight, train or bus timetables, theatre seating, cost of goods etc. The program is also an extension of that in Sect. 4.5 to show one way in which selection of one item from many can be achieved. An idea of a suitable flowchart from which, the program can be designed is shown in Fig. 5.2 which is self explanatory. The eventual program lines are added in square brackets.



(i) Flowchart "boxes"



(ii) Chart for searching through a list of ASCII numbers

Fig. 5.2 Flowcharts

```

10 REM ASCII Numbers
20 INPUT "Please quote the ASCII number for letter ....",letter$
30 FOR s=1 TO 20
40 READ char$, code
50 IF letter$=char$ GOTO 70
60 NEXT
70 PRINT
80 PRINT "ASCII for the letter "letter$;" is....";code
90 DATA A,65,a,97,B,66,b,98,C,67,c,99,D,68,d,100,E,69,e,101
100 DATA F,70,f,102,G,71,g,103,H,72,h,104,I,73,i,105,J,74,j,106

```

At line 30 the FOR/NEXT loop starts reading the DATA items in pairs into two matching variables (the first, string, the second, numeric). At each reading line 50 checks whether the letter in the data (i.e. in char\$) agrees with that entered by the enquirer (letter\$). When this happens the loop stops and processing moves to line 70 to print out the result. The data in the program can be changed easily, e.g. for timetables, prices etc.

* * * * *

The first of these two programs shows how a moderately complex one can be built up from smaller parts. Each part is designed on its own and tried out, using EDIT when the output is not as expected. Parts which occur more than once are made into subroutines. Generally it will be found that successful programming relies mainly on:

- (i) remembering the basic rules e.g. where spaces are required, use of commas and semi-colons, adding quotation marks for strings etc.
- (ii) having the list of commands and functions in the manual at hand for reference. For BASIC each one has a whole page of its own so there is plenty of information available
- (iii) learning the artful dodges which make a program do the clever tasks such as selection, pauses, act on a keypress, randomize numbers etc. Many of the latter crop up frequently in computer journals and books, not necessarily for our particular machine but usually easily adapted.

Chapter 6

BASIC – FILING

Pausing to consider that this whole book (minus diagrams) can probably be accommodated on the two sides of a single 3 inch disc (8000-type) or on one side only (9512) gives an indication of the advantages of the computer for *storing* information. With the 8000-type for example, each side of a disc holds slightly less than 180 kilobytes, equivalent to nearly 180,000 characters, or on average, 30,000 words. For the 9512 these figures are doubled. Paper may be less expensive than a disc but really not that much and apart from this the information on a disc can be changed or completely erased whereas paper can only be scrapped. The BASIC of the PCW's has been adapted to make the most of this feature so it is especially useful in business applications for example for price lists, stock lists, personnel records, address and telephone lists and in fact for almost any kind of list.

There is more in filing than meets the eye, for example with this BASIC there are three separate methods, each having its own special features. These are known as *sequential access*, *random access* and *keyed random access*, the latter with the special title of JETSAM. All three systems are perhaps easier to understand through "hands-on" experience rather than by reading about them. We start in this way with sequential access because it is the least complicated but from it most of the basic principles can be appreciated.

6.1 Sequential Access

Both sequential and random access have special requirements in that the file must not only be *opened* for the insertion or reading of data, but then *closed*. The commands involved are self-evident,

OPEN and CLOSE. For sequential files the command is followed by I for input from the file, O for output to the file.

6.1.1. Writing To a File

In this Section we set up a program for creating a file and look at methods of checking as the work progresses. The Section which follows develops a program for reading the data back from a file.

To keep the data at a minimum the file suggested is a list of 8 friends and their dates of birth. Such a small amount of data in no way detracts from the main object of understanding the underlying principles, these are the same whether there are 8 or 8000 items.

The data is recorded in the file in the order in which it is presented. We start, assuming that we have the list of 8 people and their dates of birth at hand and also a disc preferably with one side clear and formatted (Sect. 2.4).

Switch on and go through the normal procedure (Sects. 2.2 & 2.3) using System Disc No.2 (CP/M and BASIC) to arrive eventually at the BASIC prompt Ok. Change the disc for the clear one and type:

```
dir [RETURN]
or
files [RETURN]
and nothing happens because the disc is clear.
```

Next type in the following program which is not to be RUN until a copy of it is safely tucked away on the disc:

```
10 rem Friends' birth dates
20 open "O",1,"friends.seq"
30 write #1,"Jean 8-4-66","Jane 23-10-69","Joan 12-7-62","Janet 3-12-65"
40 write #1,"Tom 5-5-68","Dick 2-3-70","Harry 5-6-67","John 28-2-69"
50 close
```

Save the program under the name "bd_write" (Sect. 3.4 – bd is short for birth dates).

Type DIR again and there should be one file shown as BD_WRITE.BAS. So far so good, if anything goes wrong we still have a copy of the program safe on the disc. If the program is changed in any way then either:

```
era bd_write.bas or kill "bd_write.bas"
```

removes the old program from the disc then:

```
save "bd_write" inserts the new copy.
```

Line 20 opens the file using the *access mode* letter O (between quotation marks) because the program is *writing out* to the file. The single numeral 1 which follows is the *file number*. It can be 1,2 or 3 (or higher by arrangement). File numbers are necessary to avoid confusion if other files are open at the same time, in which case they must have different file numbers. No problem here because we are only dealing with one file. The final part of the OPEN instruction is the file name, we have chosen *friends*. The manual reminds us that it is wise to use letters and numbers only as several of the special characters (including the full stop) cannot be used.

The name must also have a file type which in this case is *seq* (for “sequential”). File types have a maximum of 3 characters.

At lines 30 and 40 the 8 separate items of data are entered into the program (not the file yet) using the command `WRITE #` (`#` is known as a “hash”). `WRITE` prints on the screen but `WRITE #` prints to a file. The 1 following is again the file number and it is followed by a list of the items to be recorded, separated by commas and enclosed within quotation marks. At line 50 the file is closed.

Next type `RUN` and we simply get the prompt back. It might appear that things have gone wrong but `DIR` tells a different story. `FRIENDS.SEQ` is now shown as a file recorded on the disc in addition to

the program `BD_WRITE.BAS`. The new file can be examined to prove all is well by:

type friends.seq

whereupon each of the entries is shown. The file is therefore safely recorded on the disc.

This has demonstrated an uncomplicated way of writing into a file, more sophisticated programs are obviously required for greater flexibility. Now follows an example in which the program places the data in an array (Sect. 5.6) before writing it out to the file. There is therefore an opportunity for manipulation of the data before filing:

```
10 rem Friends' birth dates
20 dim bd$(8)
30 for c=1 to 8
40 read bd(c)
50 next c
60 data Jean 8-4-66, Jane 23-10-69, Joan 12-7-62, Janet 3-12-65
70 data Tom 5-5-68, Dick 2-3-70, Harry 5-6-67, John 28-2-69
80 open "O",1,"friends.seq"
90 for i=1 to 8
100 write #1,bd$(i)
110 next i
120 close
```

Line 20 sets up the array under the main title `bd$` i.e. for 8 separate variables `bd$(1)` to `bd$(8)`. The data to be filed is in lines 60 and 70 and under `READ/DATA` does not require quotation marks. This is read into the `bd$` variables by lines 30 to 50 (Sect. 4.5). After line 70 therefore there are 8 variables, each containing one item of the data and in correct sequence.

At line 80 the file is opened and the contents of `bd$(1)` to `bd$(8)` are written into it by a second `FOR/NEXT` loop (90–110).

6.1.2 Reading From a File

For the file to be of any use we must be able to read it back into a program for subsequent manipulation. For this an array is again the obvious choice. The array name need not be connected with that used in the program which sets up the file (`bd$` above), writing and reading are two entirely separate operations. We choose an appropriate name, *person\$*. A file-reading program might be:

```
10 open "I",1,"friends.seq"
20 for i=1 to 8
30 input #1,person$(i)
40 next i
50 close
```

saved under the name “`bd_read`”

Line 10 opens the file again but now for reading. Lines 20 to 40 comprise a `FOR/NEXT` loop to enter the 8 items of data into the variables `person$(1)` to `person$(8)`. Note the special arrangement in line 30 with the command `INPUT #` in that `DIM` is not required to set up the array.

On `RUN`, again we get nothing on the screen but if all has gone well, the variables now contain the total data of the file e.g.

print person\$(3)

should produce Joan 12-7-62.

Our first exercises in setting up a file are completed, data has been written into one and then later on the file has been copied back into the computer. To make practical use of a file however, a few more standard BASIC techniques must be embraced.

6.1.3 Searching Within

There is little doubt that one of the more useful facilities is conspicuous by its absence, that of being able to ask the computer to find a particular entry. This in fact is not difficult to provide so, suitably modifying the programs just used, that for writing becomes:

```
10 rem Friends' birth dates
20 dim name$(8), date$(8)
30 for c=1 to 8
40 read name$(c), date$(c)
50 next c
60 open "O",1,"friends.seq"
70 for i=1 to 8
80 write #1, name$(i), date$(i)
90 next i
100 close
200 data Jean, 8-4-66, Jane, 23-10-69, Joan, 12-7-62, Janet, 3-12-65
210 data Tom, 5-5-68, Dick, 2-3-70, Harry, 5-6-67, John, 28-2-69
```

The names and dates are loaded separately into arrays *name\$* and *date\$* with commas to suit in the DATA lines which have been placed at the end of the program. Wherever they are, READ will find them.

A program for reading the file is:

```
10 open "I",1, "friends.seq"
20 for i=1 to 8
30 input #1, name$(i),date$(i)
40 next i
50 input "Name please...."; person$
60 print: print
70 for c=1 to 8
80 if name$(c)=person$ then print name$(c)" was born on "date$(c)
90 next c
100 close
110 end
```

The file items are copied into the program by line 30, using the same variable names as before. At line 50 the user is asked for the person's name whose date of birth is required. This is typed in and entered into the variable *person\$*. Lines 70–90 then search along the pairs of items until the input name and file name match. The information is printed out by the IF/THEN combination (Sect. 5.4.1) at line 80.

The programs are worth trying out. If preferred, the earlier ones may be loaded and then modified. They are next erased from the disc before the new ones are saved under the same name. Alternatively both old and new programs may be saved if the new is given a different name. The results of the exercise are for example:

```
run
Name please....? Dick
Dick was born on 2-3-70
```

Ok

This is all that is needed for an understanding of the basic principles. Many refinements can be envisaged, an essential one perhaps is that a remark on the screen should be made if the name entered is not included in the file. Such additions are more or less mini-programs on their own, each following a technique already presented in the preceding chapters. Examples in the 8000-type manuals should now make sense and show how such refinements are added.

It will be seen that in the programs suggested in the (8000-type) manuals, the file number is entered from a variable, *file%* (for % see Sect.4.4.1). This is a useful technique for programs which are constantly in use because only the line which assigns the variable needs to be edited when the file number is changed.

6.1.4 Alterations

One of the difficulties with sequential files is that additions, deletions or editing cannot be carried out directly. However the whole file can be read into the

PCW memory (e.g. by the above program minus lines 50-90), changed as required and the new version written back. An example commences at line 600 in the "phone book" program of the 8000-type manuals.

6.2 Random Access

A major difficulty with sequential access filing is that individual items cannot be brought out from the file. As we have seen, the only way is to read the whole file into variables first. Moreover, the file is not easily changed. Both these limitations are overcome in the *random access* file. Random access, as its name implies, means that any particular file item can be read or changed on demand. With sequential access we were not concerned with file item length but with random access, we are, and this has to be specified in advance.

6.2.1 A Trial Run

Let us undertake straightaway a practical example, very simplified but sufficient to show the basic principles. The file is to itemize the stock held of bird foods showing (i) the type of food, (ii) amount in stock and (iii) date of last order. Imagination is again required for application of the same principles to larger files. In this first attempt only the details of one food are recorded. Firstly we must be more precise about the terms used. In the example which follows the complete entry concerning a certain type of food is known as a *record*. All record lengths in a particular file must be the same and unless special action is taken, should not exceed 128 characters. BASIC needs to know in advance the layout or *field* of the record so before any programming can start the information to be filed must be examined to ascertain how much space is required in each record to cover all items. Also, the variable names are chosen. In the bird food example there may perhaps be 50-100 different foods but even the longest name can be written within 20 characters. For this a variable name *type\$* might be assigned. For the amount in stock (to the nearest kg) 3 characters suffice (up to 999) with the variable name *amt\$*. Similarly for the date of the last order, 6 characters and *date\$*. Note that all the variables must be string. To enter details of the layout of the records a FIELD command is used as shown in line 20 below:

```
10 open "R",2,"seed.stk"
20 field 2, 20 as type$, 3 as amt$, 6 as date$
30 lset type$="Peanuts"
40 lset amt$="175"
50 lset date$="Jan 87"
60 put 2
70 close
```

We might fancifully imagine the records to be filed on a disc as shown in Fig. 6.1. Record No.1 contains the outcome of our program, records 2 and 3 show how the FIELD instruction reserves space within

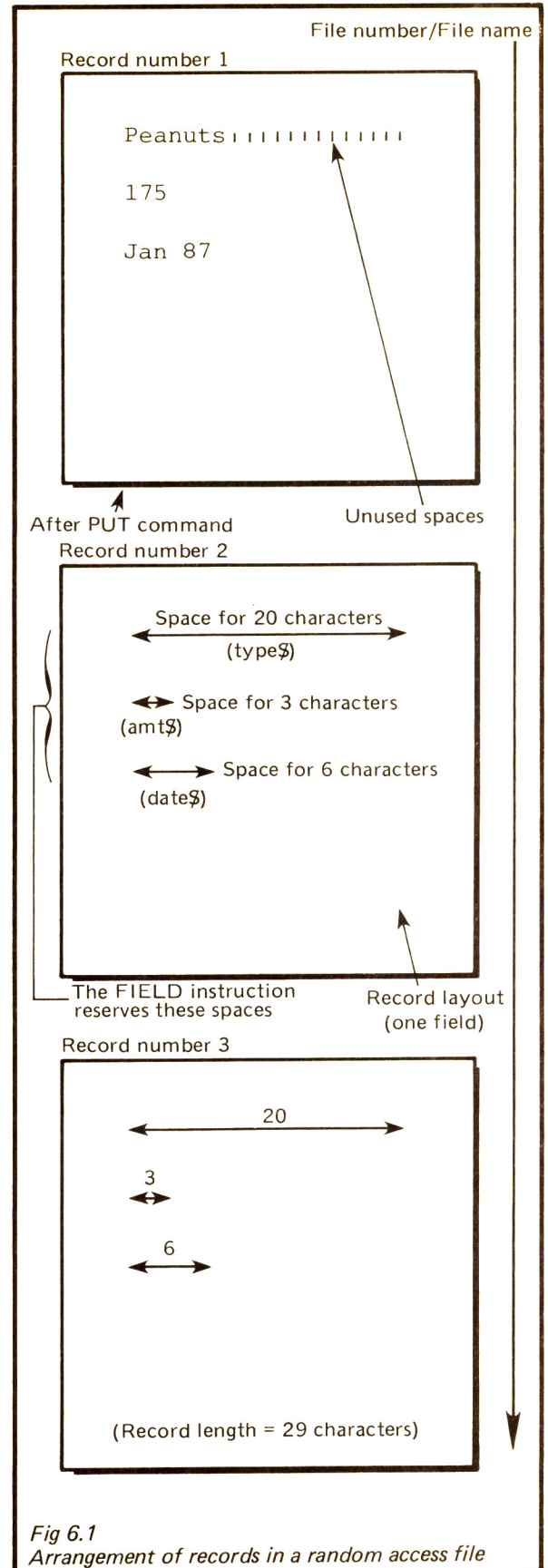


Fig 6.1
Arrangement of records in a random access file

each subsequent record. Practically any other record layout is permissible, subject to the maximum of 128 characters. Note how each record is numbered, the user must therefore be able to associate file information with its record number. Thus in the bird food file each food has a number, however we ourselves should by now be able to get over this problem – we will try later.

Line 10 opens the file using the access mode letter R and file number 2. Line 20 defines the field and note how AS is associated with FIELD. Lines 30-50 use the command LSET to insert the data in the various "field" variables. The unused character positions in line 30 (we have asked for 20 for each record but in this case only used 7) will be printed out to the right (i.e. "Peanuts" will be left justified). Other commands equally used are RSET and MID\$. Having entered the information into the field variables, line 60 files it with the command PUT, followed by the file number (2 in this exercise).

RUN copies the record out onto the disc as is shown by DIR.

To read the file entry the following program suffices:

```
10 open "R",2,"seed.stk"
20 field 2, 20 as type$,3 as amt$,6 as date$
30 get 2
40 print "Type: ";spc(16) type$
50 print "Amount in stock: "; spc(5) amt$ " kg"
60 print "Date of last order: "; date$
70 close
```

The field must again be specified as in line 20. GET is the command used to obtain a record from the file (line 30). We have only one record on file so no record number applies. At this point the information from the file has been copied into the three variables. These are printed out in lines 40-60 using the print function SPC to align the items. The print out is as follows:

Type:	Peanuts
Amount in stock:	175 kg
Date of last order:	Jan 86

Experimenting with RSET and MID\$ soon shows how to handle these.

6.2.2 A Simplified File

Clearly what has been done in the preceding Section is to find out how basically a random access file is created. One entry however does not make a file so at least programs should be produced to show how several entries are handled. Again it is merely a question of dovetailing in the various standard BASIC techniques required. It has already been mentioned that a record can only be read through its

record number. In this exercise it is assumed that the number may not be known to a user, therefore the program contains additional instructions so that it can be produced from a knowledge of the stock. We continue with the birdseed stock, still labelling the file "seed.stk". Five entries are sufficient. Writing to the file is accomplished by:

```
10 OPEN "R",2,"seed.stk"
20 FIELD 2, 20 AS type$,3 AS amt$, 6 AS date$
30 INPUT "Type of seed";t$
40 INPUT "Amount in stock";a$
50 INPUT "Month and year of last order";d$
60 INPUT "Seed Number"; sn
70 LSET type$=t$
80 LSET amt$=a$
90 LSET date$=d$
100 PUT 2,sn
110 CLOSE
```

To enter a record therefore, it is first given a file number with the INPUT commands taking in the record data. The record number is entered into the variable *sn* so that the PUT command in line 100 now has both the file number and the record number. By typing RUN and filling in the appropriate data for each record as prompted by the question marks on the screen, the first 5 records of the file "seed.stk" are entered. For those wishing to carry out the exercise fully, the data to be entered is as follows:

RECORD NO.	TYPE
1	Oat Husks
2	Flaked Maize
3	Sunflower
4	Peanuts
5	Budgerigar Mixture

STOCK (kg)	LAST ORDER
130	Nov 86
220	Feb 87
65	Dec 86
175	Jan 87
80	May 86

type seed.stk will confirm that the entries are correct

The program below which reads from the file is arranged for clarity rather than conciseness. The user needs to know the total number of records in the file (variable *c*) and the name of the seed (*s\$*):

```

10 OPEN "R",2,"seed.stk"
20 FIELD 2, 20 AS type$,3 AS amt$, 6 AS date$
30 INPUT "Number of records in file"; c
40 INPUT "Which seed"; s$
50 FOR i=1 TO c
60 READ sd$(i), num(i)
70 IF s$=sd$(i) GOTO 90
80 NEXT i
90 GET 2,num(i)
100 PRINT:PRINT
110 PRINT "Type: ";SPC(16) type$
120 PRINT "Amount in stock: "SPC(5) amt$ " kg"
130 PRINT "Date of last order: ";date$
140 CLOSE
150 DATA Oat Husks,1,Flaked Maize,2,Sunflower,3,Peanuts,4,Budgerigar Mixture,5

```

When the seed name is entered in response to the question at line 40, READ/DATA in combination with a FOR/NEXT loop (lines 50-80) finds the record number by using IF/GOTO, (Sect. 5.4.1). Because it now has the record number, line 90 is able to fill the three variables of line 20 with the record data. When new data is written into these variables, the existing data is erased. A typical result is:

```

run
Number of records in file? 5
Which seed? Sunflower
Type:           Sunflower
Amount in stock 65 kg
Date of last order Dec 86

```

Having got this program working, then alterations and additions may be made as required. Again, study of the example program of the manuals will help with ideas for improvements which are obviously needed. Nevertheless there is no need to go to great lengths in this because JETSAM which we examine next, does it for us.

6.3 JETSAM

JETSAM gives this personal computer its special aptitude for handling files, not only for the small business but also for the home. We have studied sequential and random access filing with the feeling that there is something better around the corner. So there is but it is essential to understand the *basic principles* of computer filing first. This we have done, especially with random access files for JETSAM is really an extension of them.

The manual reminds us that keyed files (as are JETSAM) are the most complex to understand, but they should not be so now that we have some experience. The same technique is followed as before, i.e. to plunge straightaway into simplified programs which we should try out for ourselves for the underlying ideas and advantages of JETSAM to be revealed. When and only when the elementary

writing and reading programs are working satisfactorily should any embellishments be contemplated. By this method, if we do happen to get into a muddle, it is possible to revert to the basic program and start again. The alternative, which is to struggle with a complex program containing all the enhancements first, is certainly inviting trouble and failure.

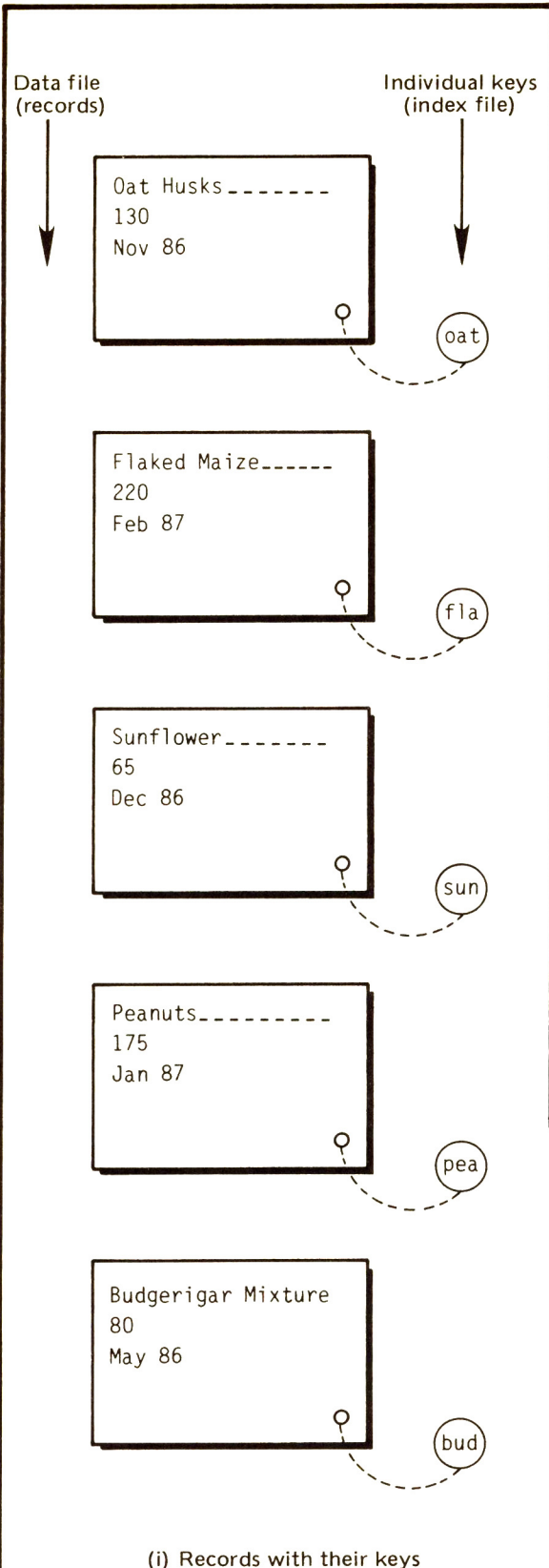
6.3.1 Keyed Files

Firstly, what is meant by a *keyed file*? Simply that access to any record in a file, irrespective of its position on the disc, is obtained by quoting a *key*. This in fact is a collection of letters, words or numbers, i.e. a *string expression* (up to 31 characters). What relevance the key has to the record itself is decided by the user. Fig. 6.2 may be of some help. It shows how the last birdseed file can be arranged under JETSAM. In this case we have chosen to make the key the first three letters of the name of the seed. It will be seen from the programs which follow that this is sufficient to read out the record associated with the key.

In (i) of the Figure, the 5 records are shown, each with its individual key. The records are contained within the *data file*, but the keys are assembled separately in an *index file*. The latter has 8 *ranks* (0-7) for the keys, which are placed in alphabetical order in the chosen rank (rank 0 in this case – see (ii)). When a rank and key are quoted to JETSAM, the appropriate record is found automatically and presented.

6.3.2 Setting Up a Keyed File

With a random access file the command OPEN both sets up and opens the file ready for use. This is not so with keyed access, an empty file must be created first and space must be reserved in the PCW memory for the index file. This latter requirement is satisfied by the single command, BUFFERS. A buffer is a section of computer memory used to hold data temporarily while other processing takes place. Each buffer allocated to JETSAM is said to require



128 bytes of memory but overall appears to need slightly more as

```
print fre(0)
31597
buffers 1
print fre(0)
31463
```

shows.

To allocate too many buffers therefore may leave too little memory for the data file, too few will result in a "memory full" message when things get going. The manual suggests that we start with 6, hence:

buffers 6 [RETURN]

reserves sufficient space for most work. The command need not be part of a program but because the memory is cleared when the PCW is switched off, it must be used when switching on again with the intention of using JETSAM. However BASIC does not seem to object to our using the command at the beginning of a program to counteract the inevitable lapses of human memory. Once BUFFERS 6 has been entered further similar entries do not reserve more space, this only occurs if a greater number is entered.

To create a keyed file the command is not unexpectedly, CREATE. Before using CREATE names must be chosen for the data and index files. Let us use "seed.lst" (for "seed list") and "seed.ind" (for "seed index") respectively. A file number must also be allocated as for random access. Hence for our purpose:

create 3,"seed.lst","seed.ind",2 [RETURN]

sets up the empty file. We happen to be using a file number of 3 here and the mysterious 2 at the end of

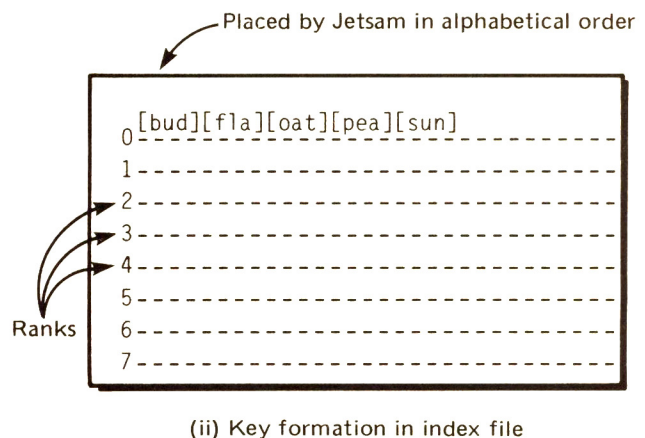


Fig. 6.2 Data and index filing in Jetsam

the instruction has to be there but it has no significance for us at present. Again this instruction does not need to be in the file writing program. So, with the PCW set for BASIC, the Ok prompt showing and with the spare disc in situ:

```

buffers 6
create 3,"seed.lst","seed.ind",2

```

gets an empty keyed file ready. DIR shows that it has its place on the disc as follows:

```

dir
SEED .LST SEED. IND
Ok

```

6.3.3 Writing To the File

A minimum program which enters data and keys is:

```

5 BUFFERS 6
10 OPEN "K",3,"seed.lst","seed.ind",2
20 FIELD 3, 20 AS type$,3 AS amt$,6 AS date$
30 INPUT "Type of seed";t$
40 INPUT "Amount in stock";a$
50 INPUT "Month and year of last order";d$
60 LSET type$=t$
70 LSET amt$=a$
80 LSET date$=d$
90 result=ADDREC(3,2,0,LEFT$(t$,3))
100 PRINT result
110 CLOSE 3

```

(It is suggested that this is typed in and also saved). The file has to be opened as in line 10, note the "K" for keyed files.

Lines 20-80 are identical with lines 20-90 (minus line 60) of the random access program of Sect. 6.2.2, except that we happen to have used a file number 3 instead of 2. Things change however at line 90. For random access the command PUT is used to enter the data but for JETSAM the key must also be entered and for this the function is ADDREC (add record+key). This however is not used directly but generally as shown, i.e. following "result = ". This is

because ADDREC generates a number to indicate progress of the writing, usually a 0 when all has gone according to plan. Incidentally this is a reminder that a function does not need to follow directly on a line number, BASIC recognizes one wherever it is.

ADDREC is backed up by the information it requires in brackets, i.e. file number, the mysterious 2 again, then rank number followed by the chosen key. In this program the rank number is 0 and the key, which is the first 3 letters of the seed name, is held in the variable t\$ (see Sect. 4.6.3 for LEFT\$).

Line 100 has been added simply to show what value ADDREC has placed in the variable *result*. Finally, and this is very important, the file is closed (line 110). Note that for keyed files CLOSE must be followed by the file number. If closure is omitted, nasty things can happen, so beware!

It is suggested that the 5 records from Sect. 6.2.2 are entered by running the program (the record number does not apply). They can be inserted in any order and will give us something to play with. The 0 appearing at the end of each insertion shows that the data and key have been accepted. In addition:

type seed.lst

will also confirm that the data has been entered correctly. The large gaps in the list are of course the unused spaces reserved in *type\$* in line 20.

6.3.4 Reading From the File

A program for reading which should also be typed in and saved follows. This is a completely separate operation from that of writing and so.

- (i) if the machine has been switched off since the last use of the file then the BUFFERS command must precede the work (unless used in the program as below)
- (ii) the file must be opened and the field layout programmed as for random access files generally. These requirements are met by lines 10 and 20 below:

```

5 BUFFERS 6
10 OPEN "K",3,"seed.lst","seed.ind",2
20 FIELD 3, 20 AS type$,3 AS amt$,6 AS date$
30 INPUT "What are the first three letters of the seed name";c$
40 first$=LEFT$(c$,1)
50 code$=UPPER$(first$)+RIGHT$(c$,2)
60 result=SEEKKEY(3,0,0,code$)
70 PRINT result
80 GET 3
90 PRINT :PRINT
100 PRINT "Type: ";SPC(16) type$
110 PRINT "Amount in stock: ";SPC(5)amt$ " kg"
120 PRINT "Date of last order: ";date$
130 CLOSE 3

```

Lines 40 and 50 take care of the fact that the user may type in the first letter of the three in lower case (Sect. 4.6.3). In line 60 again we see a function on the right hand side of an equals sign. This is SEEKKEY (seek the required record via a given key) which needs to be followed by the file number, a 0, the rank number (in this case, 0) and the key (here contained within the variable, *code\$*).

Line 70 is not essential but it does remind us that SEEKKEY also returns a number indicating success (0) or otherwise of the operation.

As with random access, GET followed by the file number brings out the data from the file into the 3 string variables of line 20. The file is closed at line 130.

This completes the program. In use, following RUN, line 30 is answered and on [RETURN] the record is displayed, typically as follows:

```
run
What are the first three letters of the seed name? bud
0
Type:          Budgerigar Mixture
Amount in stock: 80 kg
Date of last order: May 86
Ok
```

Do remember to save both these writing and reading programs, they will be very useful as the basis for further experimentation. If things happen to go wrong when adding facilities or making modifications to the programs, it is then possible to return to these basic ones and start again.

6.3.5 Gilding the Lily

As has been emphasized before, the programs of the preceding Section are basic, i.e. they do the job but with no frills attached. The latter can be added to the heart's desire using one or more of the BASIC commands and techniques discussed in Chapters 4 and 5. As a single example, the reading program does not cater for an attempt to read with a key which is not included in the file. The operative commands in this case can either be IF/THEN or WHILE/WEND using the 0 or other number returned into the variable *result* by SEEKKEY, e.g. "IF it is not a 0, then do something" or alternatively "WHILE it is a 0, carry on". Using the former, change line 70 to:

```
70 if result <> 0 goto 135      and add:
```

```
132 end
135 print
140 print ""c$"" is not in the file"
```

This gives a print out typically as follows:

```
run
What are the first three letters of the seed name? mon

'mon' is not in the file
Ok
```

JETSAM contains a host of special facilities for handling files and for analysing record data as a glance at the range of SEEK.... functions alone shows. These can all be tried out on our basic file or any other as required.

6.3.6 Making Changes

Most changes are clearly, to add, delete or change a record or key. Firstly when a file has been set up with more than a few records, the last thing we want is to lose it. There is a danger of this when changes are being made as the manual clearly shows. The safety precautions are:

- (i) make a copy of the file as a back up in case of disaster
- (ii) use the function CONSOLIDATE after changes have been made. This ensures that all new data is written onto the disc. CONSOLIDATE is another function which returns a 0 on completion.

To *add* a new record is simple. Call up the file's writing program (as in Sect. 6.3.3) and fill in the data at each prompt in the normal way.

To *delete* a record, delete its key.

* * * * *

Finally it must be admitted that JETSAM is not without its complexities. Certainly it provides useful additional filing facilities compared with many personal computers. However, unless the system is constantly in use so that the user continues to be *au fait* with its principles, efforts may frequently come to naught. Happily plenty of suitable alternative software is available.

Chapter 7

LOGO

It surely cannot have escaped the notice of 8000-type owners that in the manuals, whereas BASIC occupies a whole book of some 370 pages, LOGO is dismissed in a mere 22. Yet LOGO is certainly of no less importance. LOGO was developed in the USA in the late sixties as a language most suited to a child's patterns of thought and the way he or she works. In contrast other computer languages tend to impose on us their own way of doing things. As an example, when children have the LOGO turtle on the screen, they can decide for themselves where the turtle should go next and can cause this to happen immediately, no programming is required. By experimenting further, children enter the world of programming in a meaningful way. Once having found that the computer is not such a difficult monster to handle, children are more likely to accept the beast and move on from drawing limitless patterns and shapes to exploring more serious programming. These last chapters therefore veer slightly away from our expressed intention of only providing a back up to the manuals in that we will also explore new ground. From this the reader will be able to appreciate the basic principles of LOGO and happily go on from there to what might be described as computer thinking rather than doing. LOGO is a shortened form of the Greek *logos* (word or reason). It is not an acronym as in BASIC.

DR LOGO presents many present day home programmers with a powerful challenge. Gone are line numbers and in their place are *named procedures*. With us also are *turtle graphics*, probably the best known feature of LOGO but to the serious programmer needing to stretch the imagination, much lies behind the so-called turtle and this can involve endless hours of exploration. "Fifth generation" computers are those at present being developed in which "artificial intelligence" plays a leading role, LOGO is capable of giving us a *tiny* peep into this future. But for those with little or no experience, take heart, we will start the easy way with turtle graphics. These should present little difficulty but if they do we should be ashamed because they are used by children. From there we will move gently into the more serious aspects of LOGO programming but always with the idea of getting started rather than trying to become an expert in only a chapter or two.

Because children are a special part of the LOGO community, its error messages are as "friendly" and helpful as one could find among all the micros. In view of this we can leave error messages until the times they arrive (which is often) when it will be found that the meaning is reasonably clear. The manuals are also helpful in listing suggested actions.

7.1 The High Resolution Screen

So far we have been mainly interested in character *size*, i.e. how many characters form a line, how many lines fill the screen. While not essential, it is certainly useful to have some idea of how characters are displayed. Each character occupies the same area on the screen whether a mere full stop or a capital M. This area consists of over one hundred tiny dots, for example 16 rows of 8 (personal computers may differ in this respect). Characters are formed by illuminating the appropriate dots for the pattern required but no others. We are now interested in these tiny dots which cover the screen, generally known as the *high resolution* screen. The dots are numbered and as an example of this the maximum area of screen in which the turtle can move is approximately as shown in Fig. 7.1. This area therefore contains 720×520 (over 370,000) separate dots and we can pick out any single one of them. (But we note from the manual that the smallest distance the turtle can move is two dots). This is all we need to understand about the high resolution screen at present and there will be plenty of practice in using it in the next Section.

7.2 Turtle Talk

Section 2.3 shows the moves necessary for setting up the PCW for LOGO. However, as a brief reminder, from the prompt A> on the CP/M screen:

8000-type – change the disc for System Disc No. 4, then type:

submit logo [RETURN]

and the LOGO file will be loaded from the disc, eventually giving us its own prompt, a single question mark in the top left-hand corner.

9512 – the LOGO file is on the CP/M disc so it is only necessary to type:

logo [RETURN]

To get the turtle to join us in this escapade, type:

cs [RETURN] (clear screen)

and in the centre of the screen there appears an arrow head pointing upwards. *This is the turtle* though what relationship it bears to the reptile is anybody's guess.

7.2.1 Graphics v Text

At this stage LOGO has presented us with a "split screen", the upper part for graphics, the lower for text. This is suitable for most work with the turtle but

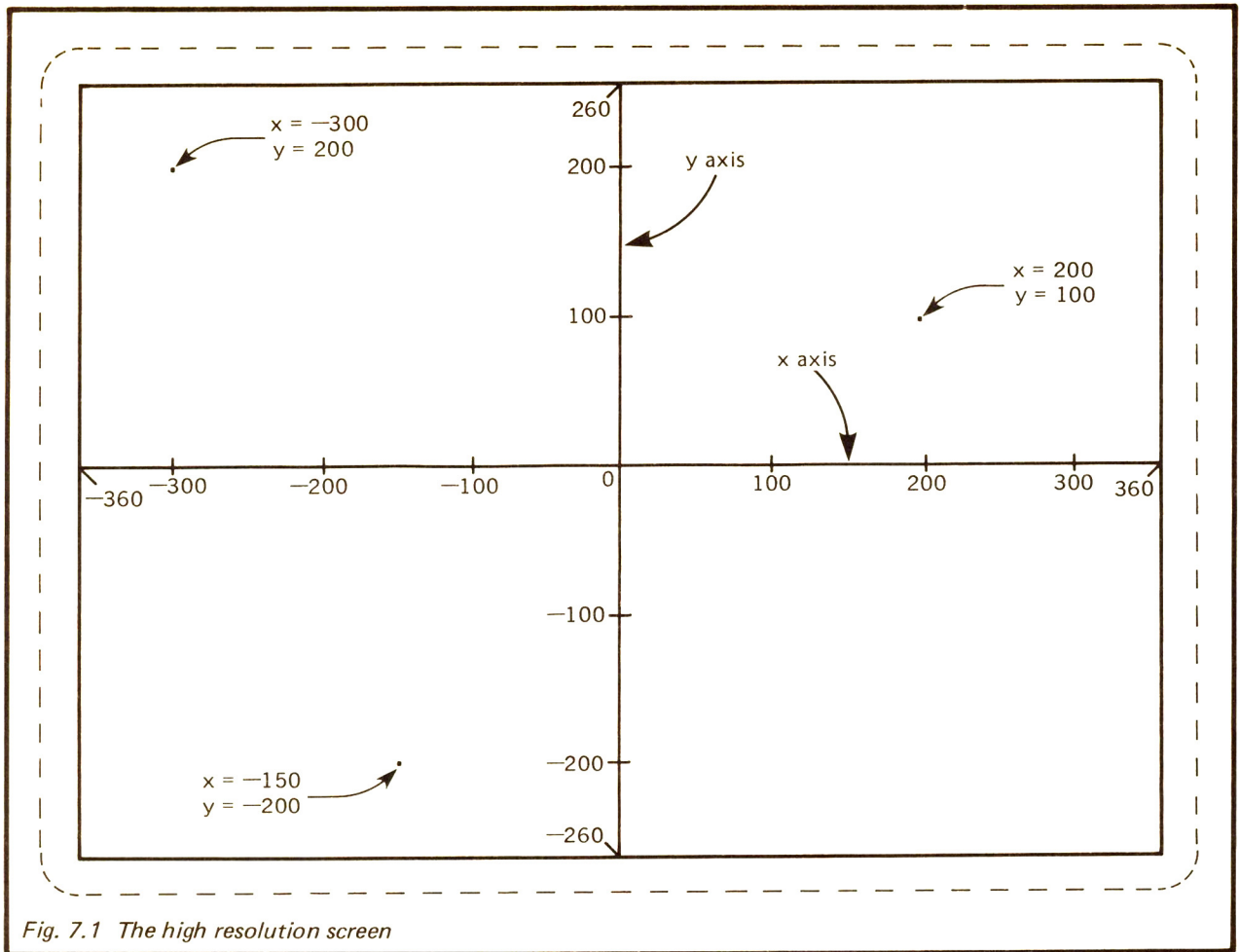


Fig. 7.1 The high resolution screen

there will be occasions when a different arrangement is better. In the standard screen, graphics occupies some 16 lines and text, 10, ideal for watching both the turtle and the input instructions together. Making a change is simply effected by the single primitive SETSPLIT which controls the number of *text* lines. To increase the graphics screen at the expense of the text screen use, say, SETSPLIT 5 which sets the text screen at 5 lines, leaving the graphics screen at 21. Equally the graphics screen can be reduced for a larger text screen area. The limits are taken over by:

fs for full graphics screen
ts for full text screen.

Return can be made to the standard screen at any time by entering SS (standard split).

7.2.2 Moving the Turtle Around

Just to see how the turtle moves type:

```
cs [RETURN]
fd 150 [RETURN]
```

(after this we will omit the [RETURN] except on occasions when a reminder is useful).

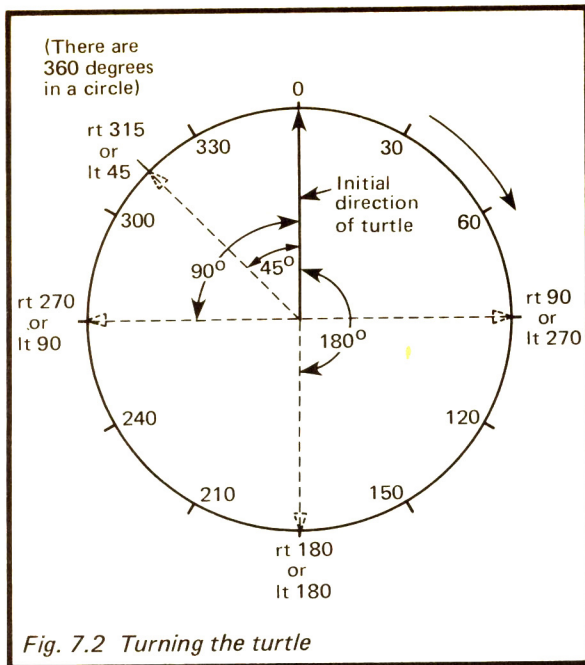
The turtle moves up the screen a distance of 150 dots or "picture elements" (known in computerese as *pixels*). CS is known as a LOGO *primitive* i.e. a primary instruction. All LOGO programming is carried out with these. The number following the primitive FD is called its *input*. Right from the start it is essential that we appreciate just how important spaces are to LOGO. In the case of a primitive for example, there is always a single space separating it from the input. Failure to observe this rule brings an error message, probably "I don't know how to...". At this stage, when one of these arrives, simply re-type the instruction.

We can turn the turtle in any direction to the left or to the right. The input is in degrees, with Fig. 7.2 as a reminder for those who have long forgotten their school geometry. The primitives are LT and RT with the input indicating the number of degrees through which the turtle is to turn.

Let us start again (with [RETURN] after each entry):

```
cs
fd 150
rt 90
```

(remember the 150 is in pixels, the 90 is the number of degrees of turn).



The turtle has moved 150 upwards and turned its head to the right. Now add:

fd 150

and the turtle moves forward *in the direction in which it is pointing*. Clearly the further addition of;

**rt 90
fd 150
rt 90
fd 150**

completes a square of side 150. At this point we may consider that the turtle spoils the symmetry of the figure, so:

ht (hide turtle) removes it
st (show turtle) brings it back
cs clears the screen.

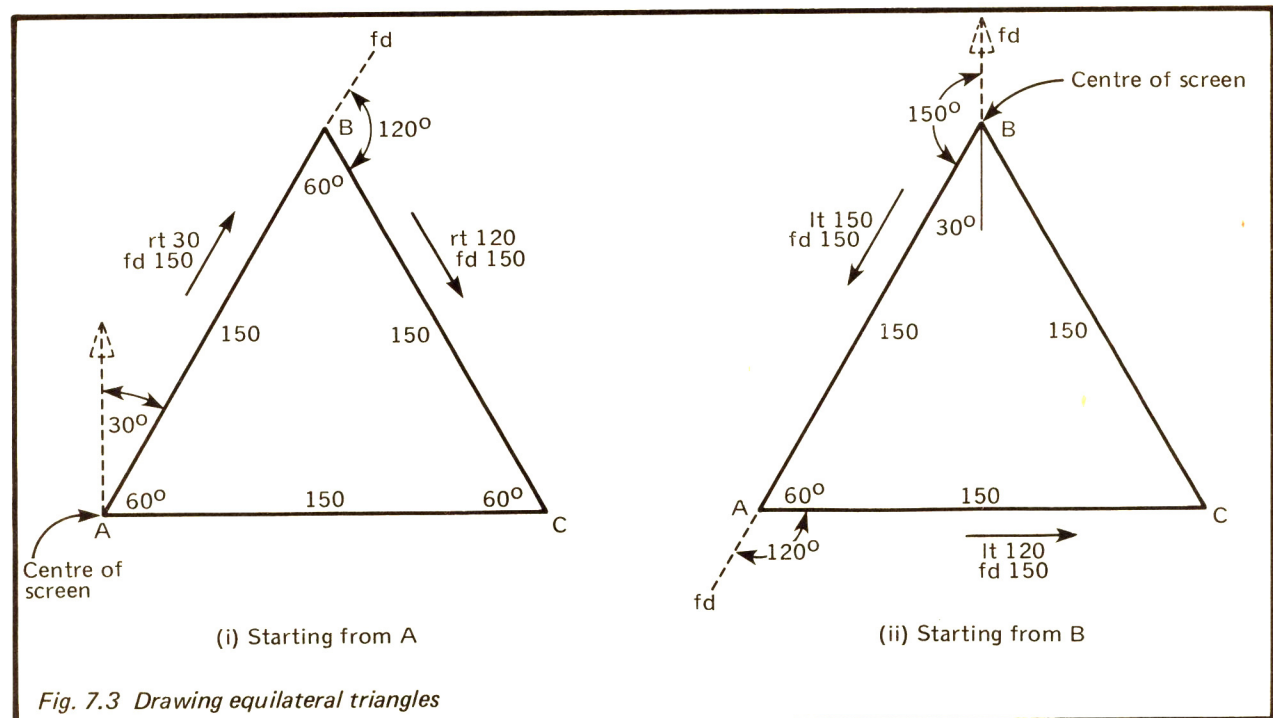
Each instruction has been entered and followed by [RETURN] so that the separate turtle movements are shown. The computer is in the *direct* or *immediate* mode which, as with BASIC, implies that nothing is committed to memory after [RETURN] is pressed. Hence editing is only possible before [RETURN] but this need not be entered after each instruction as we have done so far. Several instructions can be entered in succession first and then on [RETURN] the turtle moves accordingly. The instructions must be separated from each other by single spaces. This is demonstrated in the triangle experiment which follows.

The square is the least complicated. Let's have a go at an equilateral triangle (all sides equal). Fig. 7.3 shows the approach.

Starting from A draws the triangle in the right hand half of the screen. Such a triangle has equal angles of 60 degrees. FD goes straight up so as the Figure shows, the turtle must first be turned by 30 degrees to the right. Then FD 150 to reach B followed by a right hand turn of 120, and again at C, so:

rt 30 fd 150 rt 120 fd 150 rt 120 fd 150 ht [RETURN]

does the job. We could have used HT first. But for the next experiment, show the turtle.



Starting from B however we find that things go wrong. LT 150 followed by FD 150 as shown in (ii) of the Figure, loses the turtle. In fact it has disappeared into the text screen below. As Sect. 7.2.1 shows, the text screen size can be changed at will but at this stage let us go for a full graphics screen for which the primitive is FS. Accordingly type in:

```
fs cs lt 150 fd 150 lt 120 fd 150 lt 120 fd 150 ht
```

to produce the triangle with its apex at the centre of the screen. Now it is possible to see how children can improve their geometry in an interesting way by using the turtle.

7.2.3 The Turtle Goes Straight

A useful exercise, designed to contain most of the turtle graphics primitives and so should not be skipped over, follows. The manuals explain the meanings and use of these primitives, we put them

```
cs rt 90 fd 30 bk 60 fd 30 setpos [-100 250] seth 90
fd 30 bk 60 fd 30 setpos [-160 100] fd 120 bk 120
setpos [-200 0] fd 30 bk 60 pu setpos [0 200] pd rt 90 fd 50
bk 50 lt 90 fd 200 setpos [0 -50] setx 200 sety 0 ht
```

The result should be as in Fig.7.4(ii). This was printed out by [EXTRA] then [PTR] with it on a PCW 8256.

Graph paper is useful for setting out any particular design. The graticule should be suitable for working with the numbers of Fig. 7.1, for example, 10 squares to 2cm or 1 inch. The x and y axes are first scaled off as in Fig. 7.4 (i) and the design drawn. From this it is straightforward to draw up a turtle movement list as above. Note that there is often more than one way of making the same move.

The only turtle graphics primitives not used are PE, TF and TOWARDS [x y]. The first is useful as in pencil drawing when erasure is required, the last is called into action for example, when the angle for turning the turtle is not known and finally TF (for turtle facts) tells all about the turtle at its present position. As an example, when the Z is completed, TF gives:

```
[200 0 90 PD 1 FALSE]
```

In order, the six items of data represent:

- (1) x coordinate
- (2) y coordinate
- (3) heading
[in this case to the right therefore 90 (degrees)]
- (4) whether pen up or pen down
- (5) the current "pen"
(indicates the colour. We have only black, therefore always 1)
- (6) TRUE or FALSE for st and ht
(i.e. whether the turtle is showing or not).

to use. Try this first to see what happens, afterwards we will look at the principles of design. The use of SETPOS may be clearer by referring to Fig. 7.1. As an example, to move the turtle from anywhere to the point shown in the top left hand corner (x=-300, y=200), the instruction becomes:

```
setpos [-300 200]
```

note the single space separating the primitive SETPOS from its input, the single space separating the two items in the input and that the latter is contained within square brackets. Several instructions can be entered before [RETURN] is pressed, although perhaps one by one is better for demonstrating turtle activity. If an error occurs, try moving the turtle back to a position before the error by using BK (with LT or RT if necessary). Note that with BK, the turtle moves backwards i.e. it does not have to be turned round.

The choice of the two letters A and Z is perhaps obvious. Using the curvaceous B,C or D for example would have made life considerably more difficult. The production of curves is delayed until later.

7.2.4 Repeat

A primitive which is not strictly a turtle graphics one but has many uses in this activity is REPEAT. It also finds its way into many other LOGO operations. If a set of instructions is enclosed within square brackets, this is known as a *list* and a list preceded by REPEAT n is repeated n times. This reduces the instructions for the square of Sect. 7.2.2 to a single overall one only:

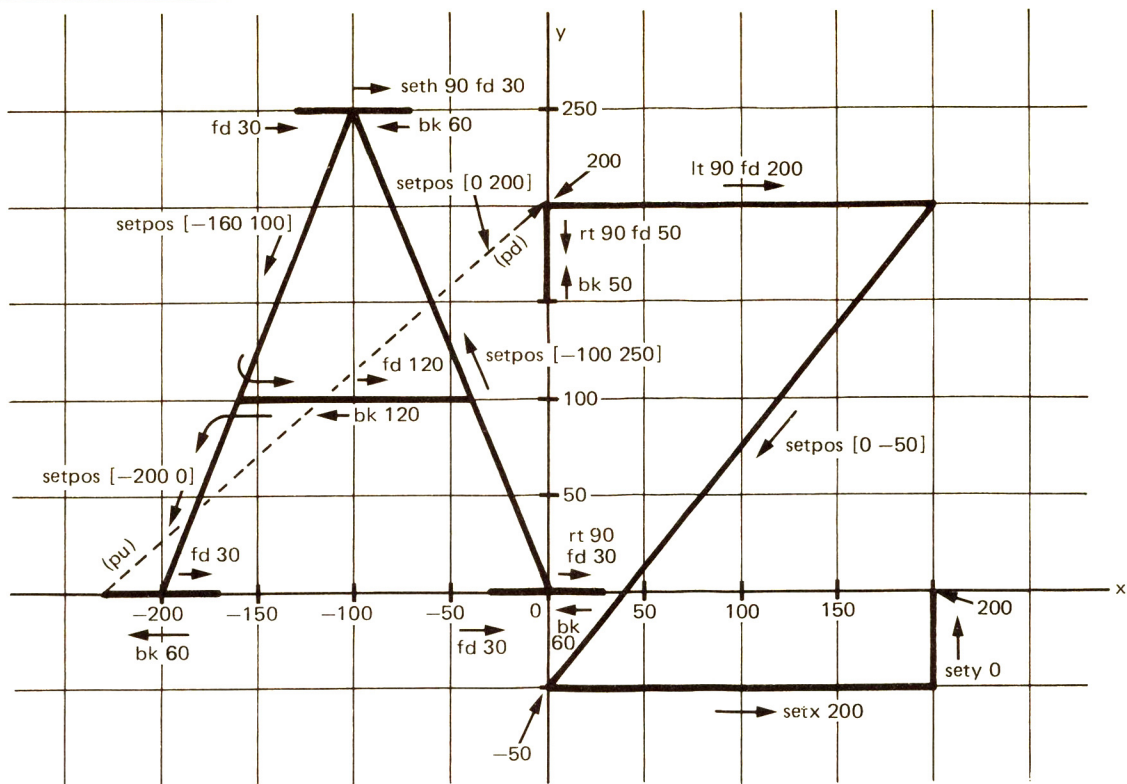
```
repeat 4 [fd 150 rt 90]
```

REPEAT therefore is similar to the BASIC FOR/NEXT loop (Sect. 5.3.1) but in fact is less complicated. With it much can be done e.g.:

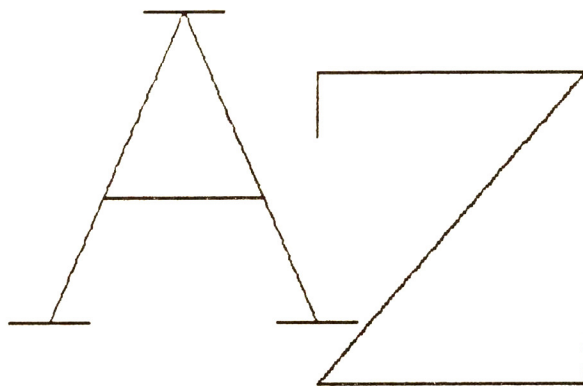
```
cs repeat 360 [fd 3 rt 1]
```

takes the turtle round in a circle of 360 moves at one degree to the right each time. However with the standard screen the turtle disappears downwards into the text. It emerges later to complete the circle (much later as it seems that when wandering within the text screen it slows down).

This can hardly be called good programming so reduce the text screen first, also move the circle to the centre:



(i) Design using graph paper



?

Drive is A:

(ii) The screen version

Fig. 7.4 Turtle tracks

```
setsplit 2 cs [RETURN]
pu setx -180 pd repeat 360 [fd 3 rt 1] ht
[RETURN]
```

The circle is not smoothly drawn as with a pen because, as seen in Sect. 7.1, the line is not continuous but jumps from dot to dot. Hence the higher the resolution of a screen, the smoother the line becomes. Incidentally, if we are unhappy about the *shape* of the circle, this is the fault of the screen, not the turtle.

7.3 Printing

Temporarily the turtle is left to his or her own devices for we now concentrate on text. Certainly printing in LOGO is a less sophisticated affair than is provided by BASIC. Essentially there are four different printing primitives with slight variations between them. (N.B. use TS for full text screen).

PR – this prints the input(s) it is given on the screen. For a single word the opening quotation marks suffice e.g.

```
pr "today [RETURN] results in:
today
```

pr "today is results in an error message because there is more than one word. It is better to use the recommended square brackets:

```
pr [today is] prints out correctly. This
primitive executes a carriage return after each input
so we get:
```

```
pr [today is] pr [the beginning]
today is
the beginning
```

The primitive TYPE does not insert a carriage return:

```
type [today is] type [the beginning of the
rest of my life]
today is the beginning of.....
```

here we have hit another snag, the two inputs although printed on the same line, have no space between them. It seems that there is little we can do about it because if a space is added after *is* or before *the*, LOGO kindly removes it. Happily the next Section shows a way out.

Note that in both cases the square brackets have been dropped on print-out.

Generally used for variables (see Sect. 8.2) is the primitive PO (print out). This reveals itself in a practical example in the Section quoted.

SHOW is different from PR and TYPE in that it does not discard the brackets e.g.:

```
show [today is] results in [today is]. This
keeps lists in list form.
```

This is the same screen as used for BASIC so as Sect. 4.2 shows, each line accommodates 90 characters i.e. in screen terminology there are 90 *columns*. The maximum number of *lines* available (full text screen) is 31. To move the cursor to any position on the screen the primitive SETCURSOR is used. It is followed by the column and line numbers in square brackets. Try the following:

```
ct setcursor [10 19] pr [This year] setcursor
[25 22] pr [Next year] setcursor [40 25]
pr [Sometime] setcursor [55 28] pr [Never]
```

The work is printed out starting at column 10, 19 lines down as below:

```

This year
Next year
Sometime
Never
```

There is an inherent difficulty in this which soon makes itself painfully obvious. An error made in typing the instructions and data may not be noticed until print out. It is then too late to correct except by re-typing it all. LOGO has "procedures" which overcome this difficulty. With very little additional effort the input is so arranged that it can be edited and printed out as many times as we wish. "Procedures" are discussed in Sect. 8.1.

Text is sent to the printer by first entering

copyon

When the bail bar on the printer is operated (or [PTR]), the printer "buttons" appear at the bottom of the screen. Any to be changed are selected by cursor and altered by the [+] or [-] keys. With the 8000-type the one we may require at present is "High quality" in preference to "Draft quality", so highlight the latter and press [+]. [EXIT] returns to normal screen. For the 9512 some experience with the printer "buttons" can be gained by making changes between "High, Medium and Low Impression", effected in the same way. The printer then copies each entry on the screen when [RETURN] is pressed. The facility ceases on entering

copyoff

7.3.1 Enter the Space

A difficulty arises in the previous Section of dividing the output of two TYPE instructions by a space. This is easily overcome by introducing the primitive CHAR which when followed by an ASCII number brings the character associated with that number into play. The ASCII number for the *space* is 32 so the earlier problem is solved by:

```
type [today is] type char 32 type [the
beginning...]
```

and the space we needed arrives.

Chapter 8

LOGO ON THE MOVE

Playing with the turtle is all very well but little serious work can be done until we are conversant with what in BASIC would be called the *programming mode*. In LOGO it is simply *procedures*. These are LOGO's way of holding lists of instructions in the memory, they can then be saved, recalled and edited or erased. A procedure can call on other procedures even including itself and this is how a program is built up, i.e. a single overall procedure which calls others as required.

8.1 Procedures

A procedure is known by its name which must not commence with a number nor be that of a primitive. Disobedience immediately produces an error message. Suppose the last program for drawing a circle and which was in the direct mode, is needed more than once. To make it into a procedure it is preceded by the word **TO** followed by the name. When the instructions have been entered LOGO is advised that the procedure is completed by **END**. So, choosing the name "circle", we first enter:

to circle

Knowing that the items of a procedure are now about to be entered, LOGO changes the prompt **? to** **>** which stays until the end. If all goes well the advice is received "circle defined". Defining *circle* from the beginning therefore:

```
?to circle
>setsplit 2
>pu setx -180 pd repeat 360 [fd 3 rt 1]
>ht
>end
circle defined
?
```

This procedure (circle) is now held in the memory and is called into action by merely typing the name, so:

circle [RETURN]

puts the turtle through its paces and

cs circle does it all again.

Two other frequently used procedure primitives (all are shown in the manuals) are:

er "name erases the procedure called "name"
po "name prints out the procedure called "name".

Note especially that for a procedure to be recognized as such and to have its instructions carried out, it is *not* preceded by either a colon or by

quotation marks. Accordingly when LOGO meets any name which is not a primitive and has no punctuation marks attached, it searches for the procedure and runs it.

8.1.1 Editing

HT is still effective from the previous run so on running *circle* for the second time we find that the procedure needs to be edited. Type in:

ed "circle

(note that opening quotation marks are required – these are necessary to avoid the procedure being run).

The screen clears and the procedure is displayed for editing. In front of the third line add **st** in the accustomed manner (i.e. place the cursor over the **p** of **pu** and enter **st** space).

Next type **[EXIT]** (or **[ALT] + C**) to indicate that the new version is to be stored and the superseded one scrapped. Running this again now has the turtle showing until the circle is completed.

To abandon an edit use **[STOP]** (or **[ALT] + G**) and the procedure remains as it was.

Here is one which has good prospects of being typed in wrongly and therefore need editing. It is another example of the use of **SETCURSOR** (Sect. 7.3) and spreads out Elizabeth Wordsworth's delightful poetry as an example:

```
to ew
ct
setcursor [10 10] pr [If all the good people were clever,]
setcursor [20 13] pr [And all clever people were good]
setcursor [30 16] pr [The world would be nicer than ever]
setcursor [40 19] pr [We thought that it possibly could.]
end
```

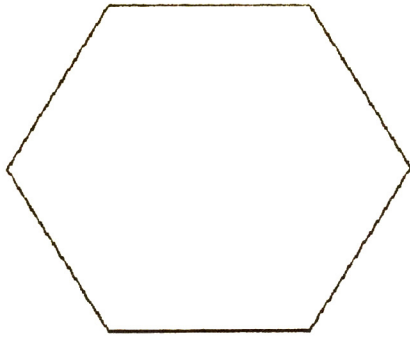
This prints the work out starting at column 10, 10 lines down. The positioning of the lines follows from the **SETCURSOR** inputs.

8.1.2 Polygons

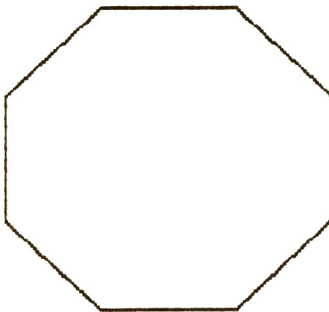
With only this small experience of procedures it is now possible to move the turtle in such a way as to further understand its educational properties. Suppose we wish to study polygons (many sided figures). Let a polygon have *n* sides, it therefore has *n* angles. We have probably forgotten this but,

sum of the internal angles of a polygon of *n* sides
$$= (2n - 4) \times 90 \text{ degrees.}$$

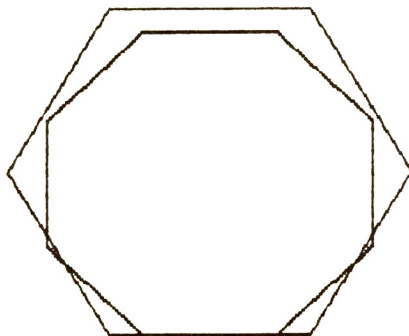
From this a general formula can be developed but



(i) Hexagon



(ii) Octagon



(iii) Mix

Fig. 8.1 Polygons

first a figure must be chosen for d , the distance the turtle moves each time. The polygon should be centrally placed so we use `setx (d/2)` to shift the starting position, then `lt 90` to point the turtle in the direction of the line, thereafter:

```
repeat n [rt (quotient 360 n) fd d]
```

to complete the polygon. (*Quotient 360 n* is the LOGO way of dividing 360 by n , giving the result as an integer).

Try this out for a hexagon ($n=6$), let $d=150$ (guesswork at this stage). The procedure therefore is defined as follows:

```
?to hexa
>cs setx -75 lt 90
>repeat 6 [rt (quotient 360 6) fd 150]
>ht
>end
hexa defined
?
```

Fig 8.1 (i) shows the result of simply entering:

```
hexa [RETURN]
```

the instructions contained in the procedure have been carried out.

This can be repeated for any polygon, e.g. for an octagon ($n=8$):

```
to octa
cs setx -50 lt 90
repeat 8 [rt (quotient 360 8) fd 100] ht
end
```

to obtain the result as in Fig. 8.1 (ii).

Finally try a 120-sided polygon:

```
to circle
cs lt 90
repeat 120 [rt (quotient 360 120) fd 6] ht
end
```

and we are back to our somewhat disorientated circle, but indicating that a true circle is a polygon with an infinite number of sides.

As procedures are defined, LOGO holds them in its *workspace* which is a section of memory reserved for this purpose. When `SAVE` is used (Sect. 3.4), the workspace is copied out onto the disc. Therefore to preserve the work we have just done for future use we only need:

```
save "polygons (or any other name chosen)
```

and all procedures held are copied out. `POALL` is useful to see what these are.

Variables which are discussed in Sect. 8.2 are also contained within the workspace and therefore are saved together with the procedures.

8.1.3 Procedures Calling Procedures

It is convenient to use two of the foregoing procedures (*hexa* and *octa*) to demonstrate the build-up of a LOGO program. Suppose it is desired to print both a hexagon and an octagon together. To do this first set up an overall procedure in exactly the same way as before and bring into play *hexa* and *octa* as required. Delete the CS in *octa* first – see Sect. 8.1.1. Call the new procedure “mix”, then:

```
to mix
  hexa rt 90
  octa
end
```

does the job when *mix* is entered. As LOGO runs through *mix* it first meets *hexa* and so runs this procedure. It so happens that the turtle finishes pointing to the left so *rt 90* is required to prepare for *octa* which follows. The result speaks for itself in Fig. 8.1 (iii). In long LOGO programs the overall procedure may call in many others, using any of them as often as required.

8.1.4 Procedures Calling Themselves

Although strictly this section should be included with more advanced LOGO programming, it is here to whet the appetite of readers who either have long been BASIC addicts or those new to the game who now begin to see the fascinations that LOGO holds. We only try out a single procedure calling itself because there are many primitives yet to be understood before full exploitation of the idea is possible.

The technique involves *recursion*. The word comes from Latin, *to run back*. From this one small program it will be seen that this is exactly what happens. With most BASIC's recursion is not possible, hence for many the technique is new and to a certain extent does not come naturally. We will try out a more advanced recursion program when control primitives have been practised.

Enter the procedure “callme” as follows:

```
to callme
  repeat 6 [rt 60 fd :d]
  make "d :d-2
  callme
end
```

The third line contains a primitive not yet encountered. We can understand it from its BASIC equivalent, LET $d = d - 2$ (Sect. 4.4). *d* is a *variable* (see Sect. 8.2) and has to be given a number to start with e.g. 150, it here represents the side of a polygon. MAKE is also explained in Sect. 8.2 so next

enter:

```
ht fs make "d 150 callme
```

and on [RETURN] a delightful pattern is traced out, going on seemingly for ever. By denying the turtle its rightful place on the screen, the pattern is completed in about half the time. It can be halted in its tracks by pressing [STOP]. Many other patterns are available e.g. *rt 1* preceding the instructions in the procedure rotates each hexagon one degree.

If more practice with editing is needed:

```
ed "callme
```

clears the screen and displays the procedure. Place the cursor at the end of the first line and press [RETURN] to obtain a blank second line. Insert *rt 1* then press EXIT to return the new version to the workspace.

A hexagon is not the only figure which can be used, an octagon is equally attractive (repeat 8 [rt 45...]). But it must be admitted that all this could be done by “conventional” methods, we are merely touching on something different.

When LOGO receives the instruction to carry out the procedure “callme” it first draws a hexagon, then reduces the value of *d* to 148. It is then instructed to revert back to itself which means doing it all again but note that this “callme” is different, it has a reduced value for *d*. There is nothing in the program to stop the process so as mentioned above, it goes on.

So, we have introduced a new technique but what is the good of that if the job can be done equally well the old way? Wait for Sect. 8.5!

8.1.5 Calculations

Already a little arithmetic has crept into the discussions. The manuals list the mathematical operations available but compared with BASIC it would appear that LOGO is somewhat lacking in its range. This may be so but with LOGO it is a simple matter to design a procedure for specific calculations. The procedures once set up can be saved in a special file, so in fact increasing the range of primitives available. The idea is expanded on in Sect. 8.2.1.

This is not a book in which we should expound on mathematical functions; sufficient to say that most can be programmed from the relatively few primitives and operators available. But take heed, several are quite complex, especially the less well known trigonometric functions and those derived from series. The latter are especially suited to recursion techniques as is seen in Sect. 8.4.

8.2 Variables

LOGO needs its variables just as much as BASIC does. The two main features of a variable are its

name and the contents. LOGO distinguishes between them by:

- (i) to quote a variable *name*, preceding it with opening quotation marks
- (ii) to refer to the *contents*, preceding the name with a colon.

There are also two separate types of variable, *global* and *local*. The first is like the BASIC variable, always available but the second is restricted to a particular procedure and any that it calls.

GLOBAL VARIABLES are set up by the primitive **MAKE**. We see this in action in Sect. 8.1.4 where to set up the variable *d* and to give it a value of 150 the instruction used is **make "d 150**. As in (i) above, *d* is the variable name, hence is preceded by quotation marks. However, in the procedure, when making changes to the contents of *d* we see **make "d :d-2**, which is saying in effect, make the variable which is named *d* ("d) reduce its contents (indicated by :d) by 2. Because this is a global variable it is available to all procedures. As a final reminder therefore:

make d 150 and **make :d 150** are both incorrect.

LOCAL VARIABLES are of special importance to designers of complex systems and apparently they avoid littering up the available workspace in the memory and therefore should be used in preference. We also find them useful, especially for procedures which can be used as additional primitives. (Incidentally the amount of free workspace can be determined at any time by **NODES** and the answer is usually in the thousands. We are told that a node is not the same as a byte but in fact it is equal to about 5 of them). Local variables do make certain operations easier in that they can be entered directly into a procedure by addition to the title. Again let us take the "callme" procedure of Sect. 8.1.4 as an example. The procedure is defined with the first line not simply **to callme** but **to callme :d** so indicating that a local variable *d* is to be used. Subsequently when the callme procedure is to be run, **make "d 150** is unnecessary, merely add the value of *d* to the procedure name so:

ht fs make "d 150 callme becomes

ht fs callme 150

More than one local variable can be entered in this way. Take as an example the calculation of the volumes of cylinders. Of course we all remember the formula $\pi r^2 h$ where *r* is the radius of the base and *h* the height. We use *r* and *h* for the variable names to save typing but the full names are equally

suitable. Define the procedure "cylinder" first:

```
to cylinder :r :h
  make "pi 3.142
  make "volume :pi * :r * :r * :h
  po "volume
end
```

 (* means multiply)

In this *pi* is a global variable, *r* and *h* are local. The latter cannot be used in another procedure because no contents have been allocated. **PO** prints out the value (or contents) of the variable *volume* but **pr :volume** is equally satisfactory.

To calculate a volume simply enter, say for *r*=2, *h*=5:

cylinder 2 5 and the result is displayed:

volume is 62.84

8.2.1 Our Own Primitives

Already some of the work done could have been made easier by having a stock of additional primitives, each designed to suit a particular need. In Sect. 7.3.1 for example, **CHAR 32** is brought into use to generate a single space but what if the requirement is for more than one space? The combination of procedures with their local variables makes this question easy to solve. Take first two procedures not using special primitives:

```
to down
  type [This procedure enters]
  repeat 5 [pr char 32]
  type [S P A C E S]
end
```

```
to right
  type [This procedure enters]
  repeat 10 [type char 32]
  type [S P A C E S]
end
```

The first prints "SPACES" on the sixth line below "This procedure enters" and the second inserts 10 spaces between the two but on the same line. Next let us create our own personal primitives to handle spacing requirements with *n* representing a number of line spaces and *s* a number of character spaces (the same variable name can be used for both if preferred):

```
to d :n
  repeat :n [pr char 32]
end

to r :s
  repeat :s [type char 32]
end
```

The two primitives *d* (down) and *r* (right) are then

called into action as shown for the first procedure:

```
to down
type [This procedure enters]
d 5 type [S P A C E S]
end
```

and here is a case where they are more useful:

```
to ws
ct d 12 r 20
type [Friends,] r 10 type [Romans,] r 10 type [countrymen,]
d 2 r 33 pr [lend me your ears]
end
```

for the result:

```
Friends,      Romans,      countrymen,
lend me your ears
```

Again, in Sect. 8.1.2 in showing how the turtle can be persuaded to draw any polygon, the primitive might be called "shape". It needs two local variables, *n*, the number of sides and *d*, the length of each side:

```
to shape :n :d
repeat :n [rt (quotient 360 :n) fd :d]
end
```

then for example, the procedure for an octagon reduces to:

```
to octa
cs setx -50 lt 90
shape 8 100
end
```

and for a circle:

```
to circle
cs lt 90
shape 120 6
end
```

in both cases using the home grown primitive, SHAPE. These primitives (they are procedures really) can be saved in the normal way and loaded back into the workspace as required.

Finally two simple primitives for calculations, they are for raising a number *n* to a given power:

```
to sq :n      to cube :n
pr :n * :n    pr :n * :n * :n
end           end
```

All that is subsequently required is for example:

```
sq 25      cube 18
625        5832
```

8.3 Lists

Lists are somewhat akin to sentences in everyday language. Information for LOGO is generally in the form of lists and it frequently needs to be processed. This involves changing words and sentences around, joining them up, taking out characters and words and rearranging lists within lists. The manuals show all the primitives associated with word and list processing together with examples of their actions. A little initiation in the techniques may be useful however. For example, extracting words or items from lists needs care, especially when lists are contained within lists.

[ab [c d]] is a list of two items, ab and [c d], the latter is therefore a list within a list.

[a b [c d]] is a list of 3 items, a, b and [c d]. LOGO sees the space between a and b as indicating that they are separate. Item 3 itself contains two items, c and d. An example of processing lists such as these arises from considering the staffing of an office. There are typists, clerks and messengers, both male (M) and female (F). The list is first recorded by the procedure:

```
to staff
make "list [[[typists(M) 1][clerks(M)
7][messengers(M) 4]][[typists(F) 8][clerks(F)
9][messengers(F) 5]]]
end
```

then extraction of lists for males and females can follow:

```
to males      to females
staff         staff
op first :list op last :list
end           end
```

both printing out the appropriate list e.g. for females:

```
[[typists(F) 8][clerks(F) 9][messengers(F) 5]]
```

The overall list contains only two lists (for males and females) therefore FIRST and LAST can be used to select either. Each secondary list evidently contains 3 others, each showing the duty and number of staff and any one of these can be extracted by ITEM or manipulated in any other way as desired. OP is used in the above procedures in case further procedures are to be used, it tells LOGO to hold the results for further processing. Changing OP to PR gets rid of the outside brackets.

Finally, note how in the procedure "staff" the variable *list* is built up. Each individual list (typists, clerks etc.) is enclosed within brackets. These are grouped in threes to form 2 higher level lists, each again enclosed within brackets. Ultimately these two lists are themselves combined into a single overall list, again enclosed within brackets. Hence we account for the 3 square brackets at both ends of the complete list.

For an example of extraction of items from elementary lists, we can look at the idea of assembling personal or other information into variables under the appropriate names. Suppose of the people within a firm, a staff list is required to contain certain information about each person, i.e. the department in which he or she works, the staff number, title, age and whether married or single. This list contains the information on 6 people, with 5 particular details on each, needless to say, there is no limit to the numbers. The ultimate requirement may be to assess the total numbers within certain age groups, how many are married etc.

It is first necessary to create global variables under the personal names, the contents of each variable being a list of:

item 1 – department	(dept)
item 2 – staff number	(staff.no)
item 3 – Mr, Mrs or Miss	(title)
item 4 – age	(age)
item 5 – whether married	(status)

Next, for example:

```
make "White_T.S [Sales 878 Miss 22 S]
make "Smith_M [Stores 863 Mr 49 M]
make "Jones_P.J [Office 214 Miss 25 S]
make "Brown_E.A [Accounts 59 Mrs 56 M]
make "Jones_A.K [Sales 431 Mr 27 M]
make "Davis_F.M [Workshops 402 Mr 48 M]
```

can be used to represent a few entries forming the *data base*.

To access any single item of information, 5 simple procedures are needed:

```
to dept :name
  op item 1 :name
end

to title :name
  op item 3 :name
end

to status :name
  op item 5 :name
end

to staff.no :name
  op item 2 :name
end

to age :name
  op item 4 :name
end
```

To access *all* the information only POALL is required but to obtain all the information about one person only we might use:

```
to all :name
  type [Department ..... ] pr item 1 :name
  type [Staff Number ..... ] pr item 2 :name
  type [Title ..... ] pr item 3 :name
  type [Age ..... ] pr item 4 :name
  type [Married:Single .. ] pr item 5 :name
end
```

For items 1 and 5, FIRST and LAST are also usable but it is better to use ITEM for all procedures for uniformity. The method of extracting information is perhaps obvious:

```
age :Brown_E.A
gives Mrs Brown's secret away
```

```
dept :Davis_F.M
shows where to find him
```

```
all :name
prints out the list of items for the person,
for example:
```

```
?all :Jones_A.K
Department .....Sales
Staff Number .....431
Title .....Mr
Age .....27
Married:Single .....M
```

(Note the colon in the last line. It is allowed within a print list and is preferable because with / or * LOGO pops in a space each side which we may not want).

Adding further entries to the data base requires nothing more than MAKE instructions. Deletions use the normal erasure primitive for variables, ERN e.g.:

```
ern [Smith_M] erases that gentleman's
entry.
```

The workspace contains both procedures and variables, hence the complete program and data base can be saved under a single name. Loading back in returns everything needed to the PCW. If

changes are made (for example, using the filename "staff"):

```
erasefile "staff" followed by
save "staff" removes the old and enters the new.
```

It must be emphasized that what we have done illustrates the basic idea only. More can be done by way of improvement especially when summaries of people or the information held about them is required. Remember too that the lists used here are elementary. More complex ones (i.e. lists within lists) can be processed almost as easily, for example, by using the method of the previous "staff" program followed by this latter one. We have used a few of the list processing primitives only but having done so, the method of use of the others should follow with no difficulty.

8.4 The Basics, LOGO Style

Many of the "standard" facilities built into BASIC and other languages also appear in LOGO. Some of these are outlined in Chapter 3. Others are developed in this Section in that we look at some of the facilities which are at the very heart of computing i.e. mainly loops and conditionals. It will be found that they exist in LOGO more as support to other special features rather than as programming techniques in their own right. Compared with BASIC the end result may be the same but devotees of this language may not find the transition to LOGO plain sailing. Accustomed as they are to a life of commas and semicolons they must now orientate themselves to colons, square brackets and spaces.

8.4.1 The Inevitable IF

Decision making *on our behalf* is one of the cornerstones of computer systems, hence the inevitable IF. The LOGO IF follows the usual pattern i.e. IF something happens, do this, IF it does not happen, do something else. Whereas BASIC needs also THEN and ELSE, LOGO manages with only square brackets:

```
if something happens [do this] otherwise
[do something else]
```

Omission of the second instruction list is in order. Try this, using the full instruction:

```
to try_if
pr [Press A or B]
if rc="A [pr [You pressed A]] [pr [You pressed B]]
end
```

We meet RC for the first time, it is one of the three primitives used for taking data from the keyboard. On the arrival of RC the processing halts until a key is pressed. It then reads the character corresponding to the key into the procedure either into a variable or comparison instruction (such as the IF above).

Entering try_if produces:

```
try_if
Press A or B      (B is chosen)
You pressed B
?
```

Here, in the *if* instruction, *rc* does not equal A, accordingly the second instruction list is operative. In this very simplified procedure, any letter other than A will of course evoke the same response ("You pressed B") but in fact only two choices are given on the expectation that the user will obey the instructions. The use of IF is demonstrated in several procedures which follow.

8.4.2 Looping

Looping, that is when the processing continually returns to an earlier instruction, involves two primitives, GO and LABEL. Because program lines are not numbered, LOGO needs some means of indicating the position to which processing is to return when the instruction to loop is received. To do this a separate instruction is added, LABEL followed a word to distinguish between other loops. Choosing the word "here":

```
label "here
(other processing)
go "here
```

forms the loop. A procedure embodying such a loop and also using some of the techniques studied earlier (local variables, REPEAT and IF) is:

```
to squares :side
label "next
repeat 4 [fd :side rt 90]
make "side :side+10
if :side < 260 [go "next]
stop
end
```

which when run by entering *squares 20* causes the turtle to generate a pattern of squares. The loop is controlled by the IF instruction. When *side* reaches 260, the *[go "next]* instruction no longer applies so the processor moves down to the next line which stops the program. Just as a reminder of a procedure calling on one of its friends try in addition:

```
to picture
cs ht squares
setx -260 squares
end
```

and when trying to run this we are reminded that there are not enough inputs to *squares*. This is true, *side* is a local variable in *squares* so its starting value is not defined. *Side* is therefore better as a global

variable so:

```
ed "squares then delete :side in the
procedure title and add a new line below:
```

```
make "side 20
[EXIT]
```

and now *picture* will run drawing two sets of squares adjoining. *Picture* has called *squares* twice, drawing the second set 260 to the left.

8.4.3 Pauses

As with BASIC pauses can be inserted by simply giving LOGO something to do which takes time and is not productive. Computers are relatively fast in carrying out tasks hence they must be instructed to do the same job many times over. Try:

```
to check_time
pr char 7
repeat 5000 []
pr char 7
end
```

("Pause" or "wait" cannot be used for the procedure name because they are primitives). It will be found that the time taken between the two pips is about 15 seconds. This works out to about 330 print operations (where nothing is actually printed out) per second. For *char 7*, refer back to the beginning of Sect. 4.7 which shows how in BASIC a single pip can be generated. The instruction used in *check_time* is simply the LOGO version through which the same ASCII number is brought into play.

8.4.4 At Random

LOGO includes a random number generator (Sect. 3.7.2) which has only one primitive associated with it. This is RANDOM which is entered followed by a number. The output is a random number from the range zero to one less than the input number. All LOGO random numbers are integers. As shown in the earlier Section, the generator always starts from the same point in its long list of numbers so to avoid repetition it is sometimes desirable to draw numbers from some way down the list. As an example first select a full text screen by **ts**, then:

```
repeat 20 [pr random 100]
```

will produce 20 random numbers from the range 0 to 99 as follows:

```
66 23 95 8 38 10 60 22 37 73 57 32 23 2 49 25 61 83 98 90
```

provided that this is the first use of random since entering LOGO.

For the range 1 to 100 the instruction becomes:

```
repeat 20 [pr random 100+1]
```

which shows that the random number arrangements for BASIC (Sect. 5.7) and LOGO have much in common.

If a fresh start is made by switching out of LOGO via **BYE** to the CP/M prompt **A>**, then back again via **SUBMIT LOGO**, the exercise is repeated and the same numbers appear.

So things are not quite as random as they look. Obviously some improvement is available if the generator is started somewhere down its list. LOGO has no **RANDOMIZE** as in BASIC so we must look to the "Guide to Amstrad LOGO" for a bit of help. The idea is to make LOGO produce random numbers one by one but not use them, so in fact moving down the list before **RANDOM** is used. Preferably the number moved down the list should vary and for this **NODES** (Sect. 8.2) is used. This is best explained by example. Try:

```
to try_ran
randomize
repeat 5 [pr random 100]
end

to randomize
make "c nodes
pr :c
repeat last :c [if random 2=2 []]
end
```

In the procedure *randomize* the peculiar looking instruction in square brackets is the one which uses up random numbers. *Random 2* must always be 1 or 0 (integer in LOGO means simply chopping off the decimal fraction), hence *random 2* can never equal 2. Accordingly the IF instruction is repeated *last :c* times. "c" is the number of nodes at that instant and *last* is one of the word and list processing primitives, it extracts the last letter or figure. The IF instruction must be completed so an empty instruction list [] is added. A typical result soon after switching to LOGO is:

```
try_ran
3459
73
57
32
23
2
?
```

from which it is seen that *nodes* (and therefore "c")=3459, *last :c* is 9 and the 5 random numbers start after 9 have been drawn and thrown away.

Amstrad's Guide suggestion is better still but mathematically more complicated. Used instead of *last :c* the line becomes:

```
nodes - 50 * int (quotient nodes 50)
```

which in plainer English is "nodes less 50 times the

integer of nodes divided by 50", so giving larger numbers up to 49. Note that we are not entirely out of the wood because if after switching on the same operations are used as before, then *nodes* will be the same and the degree of movement down the random numbers list will also be the same. Such an occurrence is unlikely however.

We can see random numbers in action in the next program. It is for a simple game, ideal for teaching children to use a logical approach in solving problems. The game is found in many computer books, here we call it "Numbers". The computer chooses at random a number between 0 and 100 but does not disclose it. The player enters numbers of his or her choice within the range whereupon the computer indicates whether the input number is higher or lower than the one held. Three procedures are involved:

- (i) *numbers* – which picks up a random number and organizes the game
- (ii) *mark* – which takes the input number and checks it against the secret one
- (iii) *randomize* – as developed above.

The complete program is as follows:

```

to numbers
make "tries 0
randomize
make "mine random 100
pr [] pr [I have a number between 0 and 100
in mind] pr [] repeat 250 []
pr [Try to guess what it is] pr [] repeat 250 []
pr [Type in your guess and press RETURN]
pr [] pr []
mark
end

to mark
make "tries :tries+1
make "yrs first rl
if :yrs=:mine [(repeat 5 [pr char 7]) pr [] (pr
[Got it in] :tries. Do you want to try again?
*** Press Y for yes or STOP for no.)) (if rc="y
[numbers])]
if :yrs > :mine [pr [Too high. Try again]]
[pr [Too low. Try again]]
mark
end

to randomize
make "c nodes
repeat last :c [if random 2=2[]]
end

```

The game is started by entering:

numbers [RETURN] as shown below:

?numbers

I have a number between 0 and 100 in mind

Try to guess what it is

Type in your guess and press RETURN

```

50
Too high. Try again
25
Too low. Try again
37
Too high. Try again
31
Too low. Try again
34
Too high. Try again
32

```

Got it in 6. Do you want to try again? *** Press Y for yes or STOP for no.

The procedure *numbers* is straightforward and uses a slight delay in the printing to add effect (Sect. 8.4.3). It finally calls the procedure *mark* which on each attempt increases the variable *tries* by 1. RL reads the characters typed in. RC cannot be used because it only reads one character. It follows later in the IF instructions to return processing to the procedure *numbers* if key Y is pressed. IF instructions are used to mark the try and then print out accordingly. Sorting them out takes a little patience, there are often many brackets to consider. Accordingly another look at Sect. 8.4.1 may be helpful in deciding where the individual instruction lists begin and end.

[pr char 7] generates the squeaks of delight when the player is successful.

Note that *mark* is a recursive procedure with escape from it via an IF instruction (if rc="y [numbers]).

The use of logic in playing the game arises from the fact that the number of tries need not be greater than 7. The chance of success at the beginning of the game is 1 in 100. By first entering 50 the range is halved and the chance changes to 1 in 50. By similar reasoning, if the range within which the secret number lies is halved each time instead of using a wild guess, then the number will be discovered within no more than 7 tries.

8.5 Recursion Recurs

The idea of recursion first appears in Sect. 8.1.4 but in that Section there seems to be little out of keeping with normal practice. However let us see what happens when for example, a procedure which calls itself is rearranged.

Firstly here is one unchanged:

```
to decrease :n
type [Printout No....] pr :n
make "n :n - 1
if :n=0 [] [decrease :n]
end
```

With our experience this should present no difficulty. *n* can be made any number and this decreases by one on each recursion. Ultimately the IF instruction steps in when *n* reaches zero, after this, END is effective. There is no printout therefore for *n*=0. The result of running the procedure is as might be expected:

```
?decrease 5
Printout No....5
Printout No....4
Printout No....3
Printout No....2
Printout No....1
```

Next the procedure is rearranged, now calling it "increase :*n*". The immediate reaction is that nothing can increase with an *n*-1 term built in which itself can only decrease. But wait and see. Enter:

```
to increase :n
make "n :n - 1
if :n=0 [] [increase :n]
type [Printout No....] pr :n
end
```

and now the output shows:

```
?increase 5
Printout No....0
Printout No....1
Printout No....2
Printout No....3
Printout No....4
```

a very different story. 0 has crept in and the printout numbers increase. Yet exactly the same lines have been used except for the procedure names. The obvious difference between the two procedures is that in the latter (increase), all recursion is completed *before* the program reaches the *type* line. When it does, however, all 5 items are printed out but in reverse order! The process is known as *unwinding* the recursion. Two features are peculiar to this process:

- (i) on each recursion the procedure itself changes, for example we start off with *increase 5* but the next loop starts as *increase 4* and so on. Hence in this case different procedures are involved.
- (ii) only on the last loop does the procedure appear

to reach END. Here it prints out 0. The other procedures need completion so the system runs backwards (unwinds) through these and in doing so completes each 4th and 5th line so printing out in reverse order. The numbers are each one less than is given by the procedure *decrease :n* because the first move reduces *n* by one before anything else happens.

This particular phenomenon is neither easy to explain nor to understand as are many ideas concerning artificial intelligence. Its uses cover a wide field, sufficient to mention here perhaps that it can even make solution of that *bête noire* of school mathematics, the infinite series, relatively easy.

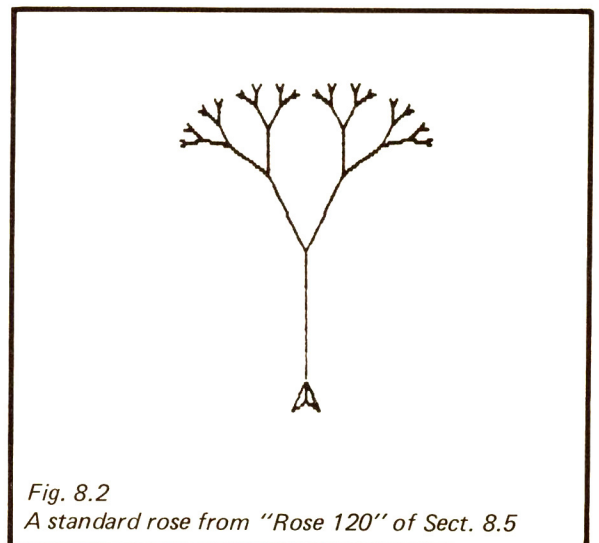
For another teaser, here is one way of drawing a standard rose tree using *multiple recursion*, a type beloved of many LOGO experts. Most exponents of the art have their own favourite tree, ours is a simplified version but it works well. The procedure is named "rose" and *h* is the height of the main stem:

```
to rose :h
if :h < 4 [stop]
fd :h rt 25
rose :h / 1.8
lt 50
rose :h / 1.8
rt 25 bk :h
end
```

STOP in the second line does not halt the program, processing stops at that point but then reverts to the calling procedure (see next Section). Fig. 8.2 shows the result of entering:

```
cs rose 120
```

(with any greater number for *h*, some of the branches disappear out of the top of the screen).



This is not an easy one to unravel. For a start, with *h* at 120, *fd h* in the third line moves the turtle vertically and from the same line is then turned 25 degrees to the right. The 4th line next calls the main procedure again but now modified to a smaller value for *h*, i.e. $120 / 1.8 \approx 67$. By continuing this recursive action the turtle plods its way round the right hand side of the figure until *h* falls to less than 4 (line 2) and the instruction STOP becomes effective. The turtle is then at the tip of one of the smallest branches.

Now, with such complex recursive action things get really difficult to understand and even more difficult to describe. It is probably better to watch the procedure in action on the screen, especially if the turtle is hidden. At least we gain a wholesome respect for the power which can reside in a very short recursive procedure.

8.6 Moving Between Procedures

A little more light can be shed on the multiple recursion procedure of the previous Section by getting to grips with the idea of processing at various *levels*. When the basic procedure is running, the system is said to be at level 1 (some say level 0). Now suppose this procedure calls in another, while the latter is running the system is said to be at level 2. Again another procedure called in changes the system to level 3 and so on. Reversion to the first (basic) procedure for example returns to level 1. In the *rose* procedure of the preceding Section therefore it is necessary to be conscious of the various levels at which the processor is working. Accordingly when STOP is encountered, then the procedure which takes the next step is the one which called the procedure containing STOP. LOGO has a system built in by which reversion to any previous level can be effected (not as with STOP which returns to the calling one only). The point to which the processing returns is indicated by CATCH, a primitive which is followed by an identifying word and the directions as to how to continue the program.

The point *from which* the program is redirected is indicated by THROW, this is followed only by the same identifying word. For example *throw "word* moves control back to *catch "word [program instructions]*. It has a similarity with *go "word* moving processing back to *label "word* except that THROW can operate between different procedures whereas GO can only work within a procedure. The program statements in the CATCH instruction are first processed normally but when control is passed back, then the statement following is executed. We try this out with:

```
to proc.1
pr [This is printed by procedure 1]
catch "try [proc.2]
repeat 600 []
pr [This is the second printing by procedure 1]
end
```

```
to proc.2
repeat 600 []
pr [This is printed by procedure 2]
proc.3
end
```

```
to proc.3
repeat 600 []
pr [This is printed by procedure 3]
throw "try
end
```

with the result on entering **proc.1**:

```
This is printed by procedure 1
This is printed by procedure 2
This is printed by procedure 3
This is the second printing by procedure 1
```

There should be little difficulty in tracing the flow of processing through these three procedures. Proc.1 prints the first message, then at the third line (*catch*) moves on to proc.2. This one takes over, prints its own message and hands over to proc.3. Again the message is printed out. After this *throw "try* moves back to proc.1 at *catch "try*, from which the final message is printed.

While these procedures are in the workspace it is worth checking on STOP. Add STOP so that the last three lines of proc.2 are:

```
stop
proc.3
end          the result from proc.1 is
```

```
This is printed by procedure 1
This is printed by procedure 2
This is the second printing by procedure 1
```

STOP has prevented proc.2 from calling proc.3 and directed processing back to the calling proc.1 at the point where proc.2 is being called.

THROW has a second string to its fiddle. THROW "TOPLEVEL moves control back to the first (or top) level, the whole program stops and the normal prompt (?) arrives. TOPLEVEL must be entered in capital letters otherwise LOGO hunts for a CATCH instruction to suit. A simple example follows:

```
to check
pr [Do you want to try again? Type y or n]
if rc="y [pr [Good. Off we go then]][throw
"TOPLEVEL]
end
```

In this particular case if *n* (or any letter other than *y*) is typed, the program is abandoned by *throw "TOPLEVEL* in the IF instruction.

Chapter 9

LOGO – DATA BASES

A *property* is something which is owned. Together with its owner, LOGO considers the combination to be a *property pair*. A property is then said to consist of two items, a *name* and its *value*.

Properties are a very important part of LOGO and in fact its own way of storing variables and procedures is in this form.

9.1 LOGO Properties

Take the simplest variable first:

```
make "x 25
```

Here x is the name (of the owner) and 25 the value of that which it owns. In this chapter we meet a range of special “property” primitives. The first is PLIST which takes the name of the property (in this case x) and outputs its property list or value. (We recall that a *list* in LOGO is any number of characters, words or numbers enclosed within square brackets). Accordingly:

```
plist "x
```

prints on the screen the property list (value) of x. The output is:

```
[.APV 25] where .APV stands for “Associated Property Value”. For a little more experience, try a variable used in Sect. 8.4.4:
```

```
make "mine random 100
```

for which PLIST reveals [.APV 66] (or some other number – see Sect. 8.3.4). Random 100 as it stands has no value until called so LOGO has done this and is waiting for the value to be used. Note that we are not *using* the variables here, merely looking at them.

A list within a list follows. This also shows that the value of a property need not be numerical, it can be a word or sentence:

```
make "time [The mouse ran up the clock]  
plist "time  
[.APV [The mouse ran up the clock]]
```

Variables, which we now see are stored as properties and indicated by .APV are themselves part of procedures. These too LOGO stores as properties but in this case the various instructions are stored as lists. On printout from PLIST, instead of .APV which refers to variables, LOGO uses .DEF. This is an indication that what follows is the DEFINITION of a procedure stored as a property under the procedure name. The next example is also an exercise in

sorting out square brackets. We use the procedure *decrease :n* developed for Sect. 8.5. On:

```
plist "decrease :n LOGO replies with:
```

```
[.DEF [[n] [type [Printout No....] pr :n] [make "n :n-1]  
[if :n=0 [] [decrease :n]]]]  
n has no value
```

Clearly most of this comprises the various instructions exactly as they were entered but each made into a list. The bracketed n following .DEF shows the local variable name added to the procedure name. If none is used, .DEF is followed by []. Because variables and procedures are distinguished by different property titles (.APV and .DEF) it is possible for a program to contain one of each under the same name.

The statement that “n has no value” is obviously true because we are examining a stored procedure which is not in use. The variable n is given a value by the user only when the procedure is called into action.

9.2 Creating Properties

The use of the property approach enables the programmer to set up *data bases* from which information in many forms can be extracted. Human intelligence relies on an enormous accumulation of information from which decisions can be made. Artificial intelligence likewise needs its store of facts. i.e. a data base. These facts are stored in the form *name, property, value*, an example being that John (the name) has the property of age which has a value of 26 years. This is the basic form on which a data base is set up. Understanding this is much easier by example. Two more primitives are needed to handle the data:

pprop (put property) – to install it in the memory
gprop (get property) – to get it out again.

PPROP must be followed by name, property, value, in that order thus:

```
pprop "John "age "26 On [RETURN] this
```

is entered into the memory as a property as

```
plist "John confirms by [age 26]
```

9.3 Setting Up a Data Base

Let us add a couple of John’s friends, Mary and Anita:

```
pprop "Mary "age "22  
pprop "Anita "age "25
```


PLIST followed by any of the names brings out that person's age. Next suppose that more data is added giving the weight and height of each e.g.:

```
pprop "John" "weight" "63kg
pprop "John" "height" "168cm
pprop "Mary" "weight" "57kg
pprop "Mary" "height" "157cm
pprop "Anita" "weight" "60kg
pprop "Anita" "height" "162cm
```

The data base is now capable of selecting any item or items of information about our subjects. It is printed out in one list e.g.:

```
plist "Anita
[height 162cm weight 60kg age 25]
```

If the need is for the value of the property only as may occur in certain programs, the primitive GPROP (get property) is employed:

```
gprop "Mary" "height
[157cm]
```

To keep track of what is already documented, GLIST (get list) is available. This outputs a list of names associated with a certain property, e.g.:

```
glist "age
[Anita Mary John]
```

Removal of any entry is effected through REMPROP (remove property), followed by the name and property, e.g.:

```
remprop "John" "weight
```

whereupon *plist "John"* confirms that only his height and age remain recorded.

To print out all the property pairs against each name, PPS is available:

pps [RETURN] with the data base referring to John and his friends gives:

```
Anita's height is 162cm
Anita's weight is 60kg
Anita's age is 25
Mary's height is 157cm
Mary's weight is 57kg
Mary's age is 22
John's height is 168cm
John's weight is 63kg
John's age is 26
```

The *apostrophe s* and *is* in each line are added by LOGO, an original and pleasing mode of expression. The fact that Anita and Mary are discussed before John is probably due to good manners.

Although all of this can be arranged by BASIC, it is clear that LOGO has a special affinity with data bases. In fact once one knows the primitives involved, the whole process of setting up a data base and reading information from it is quickly understood with only a minimum of experience. Saving a data base is straightforward, we do so in the next example.

A summary of the data base primitives follows because these are mentioned in the manuals but not explained:

pprop "name" "property" "value
adds these to the data base

plist "name" prints out property and value

gprop "name" "property" prints out value

glist "property"
prints out the list of names associated with this property

pps prints out all the property pairs

remprop "name" "property"
deletes property associated with name.

9.3.1 A Data Base Transposed

Once the data base system is fully understood it becomes possible to add to its efficacy through procedures designed for its use in the outside world, (for example, who except LOGO people has ever heard of *plist* or *gprop*?). Slightly more advanced programs which are unfortunately beyond the scope of this book, can put questions and give answers in everyday English. As an idea of this, for a data base of a family tree the questions and answers might be in the form:

Was "a" married to "b"?

Yes

Did "c" have any children?

Yes Penelope and Peter

Did "d" have any children?

No

All very nice, but we ourselves must walk before we run. What can be done however is to experiment by moving away from the *name/property/value* concept to others of our own. The data base which follows is not expected to please an expert but it does have the advantage of being easy to understand because no complicated procedures are used. All that is required of the user is a knowledge of the correct way to enter the information.

First let *name/property/value* give way to *place/link/object*.

The idea is to create a data base of towns and cities with certain information regarding them. As an example, Rome is a city where *Rome* is the place,

is_a is the link and *city* is the object. The requirements are:

- (i) to print out the total information on a particular town or city
- (ii) to summarize the places having similar characteristics.

Places (towns and cities) are recorded in the data base together with the information (object) as to which country they are in, whether on a river, at the seaside, have a port etc. The "links" used in each case are different, we will see why as we progress. It is suggested that readers may wish to copy the data base and associated procedures into their PCW's, then save all onto disc for future experimentation. Note that LOGO reserves spaces for operational uses hence the underline symbol is generally employed instead. Also SHOW can be used instead of PR when brackets are required on the lists.

First the procedures:

```
to add :place :link :object
pprop :place :link :object
end
```

```
to delete :place :link
remprop :place :link
end
```

(this simply allows us to add and delete items rather than pprop or remprop them).

```
to place :name
show plist :name
end

to summarize
pr [Cities : -] cities pr []
pr [Towns : -] towns pr []
pr [Ports : -] ports pr []
pr [Rivers : -] rivers pr []
end
```

```
to cities
pr glist "is-a
end
```

```
to towns
pr glist "is-classed-as-a
end
```

```
to ports
pr glist "has-a
end
```

```
to rivers
pr glist "is-on-the-river
end
```

Next enter the first few items of the data base. There is no difficulty in adding to it later.

```
add "London "is_in "UK
```

```
add "London "is_a "city
add "London "is_on_the_river "Thames
add "London "has_a "port
add "Liverpool "is_in "UK
add "Liverpool "is_a "city
add "Liverpool "is_on_the_river "Mersey
add "Liverpool "has_a "port
add "Rome "is_a "city
add "Rome "is_on_the_river "Tiber
add "Rome "is_in "Italy
add "Sorrento "is_classed_as_a "town
add "Sorrento "is_in "Italy
```

To be on the safe side, save the data base and procedures together in the normal way. Choosing for example, the title "places" and with a disc of sufficient spare capacity in the drive:

```
save "places
```

copies everything in the PCW workspace out onto the disc, i.e. the data base and its associated procedures are saved together. Now if the machine is switched off or there is a need to revert to CP/M, when back with LOGO again:

```
load "places restores it all.
```

(Incidentally, by typing *nodes* before and after the LOAD instruction, it will be seen how many are required.)

The display lists all the procedures but makes no mention of the data base. That it is there can quickly be proved by:

```
summarize with the reply:
```

```
Cities : -
London Liverpool Rome
```

```
Towns : -
Sorrento
```

```
Ports : -
London Liverpool
```

```
Rivers : -
London Liverpool Rome
```

Obviously the data base is very small but it is sufficient to demonstrate the system on which it works. For details about any particular place, for example, Liverpool:

```
place "Liverpool is all that is needed:
```

```
[is_in UK has_a port is_a city is_on_the_river Mersey]
```

If however the need is to compile a list of places having one particular feature only, then simply:

```
cities prints out London Liverpool Rome
towns prints out Sorrento
ports prints out London Liverpool
rivers prints out London Liverpool Rome
```

It is now evident as to why the links must be different for each object. This is because each is part of a property pair and therefore must be quoted to indicate the associated value, or in this case, object. Hence, for example, the object *port* must always be associated with the link *has_a*.

PPS which served us well in the previous “John and friends” data base is not so neatly put with this second one but then we have strayed from LOGO’s method.

Remember that if alterations are made to the data base or procedures, the new edition should be saved, so with the same disc in situ:

erasefile “places removes the old version and

save “places inserts the new one,

the progress of which can be checked by DIR at any time.

Again, not an elegant solution to the problem of setting up a data base, but it is a start and should present little difficulty in gaining a full understanding of the basic system. Data bases can have a practical use as we have seen. Alternatively, for those who wish to go further, they can provide an endless combination of satisfaction and frustration in further experimenting. Then undoubtedly the busiest primitive of all will be ED.

* * * * *

Especially with LOGO, writing a program is very much a matter of personal style. There is no magical system built in to do the job for us, all the interpreter can do is to prevent us from deviating too far from the straight and narrow LOGO path. Loose thinking is definitely not allowed and this is perhaps one of the challenges that LOGO offers. Our own expressed intention is to “get started”, hopefully this has now been achieved and the pleasure of using LOGO to the full is yet to come.

Appendix

SINGLE BYTE BINARY/ DECIMAL CONVERSION

In Chapter 1 is discussed briefly the use of 8-bit (one byte) binary codes. There it is mentioned that there are 256 *different* combinations of 0's and 1's obtainable from such a code. This Appendix shows them all in a 16 x 16 Table (A.1) with their decimal equivalents. The latter are in fact not 1–256 but 0–255. Although the Table itself is more or less self-explanatory, here are two examples:

Conversion of decimal 219 to binary:

Decimal 219 is found in the third column from the right and fifth line up. The first 4 bits (or digits) of the

binary equivalent are at the head of the column, i.e. 1101. The second 4 bits are in the column at the extreme left, fifth from the bottom, i.e. 1011. The full binary number is therefore 11011011.

Conversion of binary 00100001 to decimal:

The first 4 bits are 0010 therefore the required number is in the third column. The second 4 bits are 0001 and appear in the second line down of the left hand column. At third column, second line down is the equivalent decimal number. i.e. 33.

TABLE A.1 1-BYTE BINARY NUMBERS

1st 4 → bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
2nd 4 ↓ bits 0000	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
0001	1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241
0010	2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242
0011	3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
0100	4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244
0101	5	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245
0110	6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246
0111	7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247
1000	8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248
1001	9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249
1010	10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250
1011	11	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251
1100	12	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252
1101	13	29	45	61	77	93	109	125	141	157	173	189	205	221	237	253
1110	14	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254
1111	15	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255

Index

Commands and functions in BASIC and primitives in LOGO are given in *italic capitals*. Which language applies can be gauged from the fact that LOGO commences on page 47 and continues to the end of the book.

A	Page		Page
Access mode	37	<i>ERASE</i>	7
<i>ADDREC</i>	44	<i>ERASEFILE</i>	59
<i>ALT</i>	15	<i>ERN</i>	58
American Standard Code for Information Interchange	11	Error messages	11
Argument	11	ESC	25
Array	30	Escape sequence	25
Artificial intelligence	47	EXIT	18
AS	41	EXTRA	31
ASC	11	F	
<i>ASCII</i>	12	<i>FD</i>	48
Associated Property Value	65	Field	40
<i>AUTO</i>	15	<i>FIELD</i>	40
B		Fifth generation computers	47
BAS	19	Filename	7
<i>BASIC</i>	8	Files	7
Binary digit	1	<i>FILES</i>	37
Bit	1	Filetype	7
<i>BK</i>	50	<i>FIRST</i>	57
<i>BUFFERS</i>	42	Flowchart	32, 34
Bug	22	<i>FOR</i>	27
<i>BUTTONS</i>	9	Format	16
<i>BYE</i>	8, 60	Formatting	9
C		FORTRAN	3
<i>CATCH</i>	63	<i>FRE</i>	17
<i>CHAR</i>	12, 52	<i>FS</i>	48
<i>CHRS</i>	11	Functions	11
<i>CLOSE</i>	37	G	
<i>CO</i>	9	<i>GET</i>	41
COBOL	3	<i>GLIST</i>	66
Colon, use of in BASIC	16	Global variables	56
Column	16, 52	<i>GO</i>	59
<i>COM</i>	7	<i>GOSUB</i>	29
Comma, use of in BASIC	16	<i>GOTO</i>	27
<i>COMMANDS</i>	9	<i>GPROP</i>	65
Commands	11	H	
Concatenation	23	<i>HELP</i>	9
<i>CONSOLIDATE</i>	45	High resolution	47
<i>CONT</i>	27	<i>HT</i>	49
<i>CONVENTIONS</i>	9	I	
Copying	9	<i>IF</i>	29, 59
<i>COPYOFF</i>	52	Immediate mode	49
<i>COPYON</i>	52	Index file	42
CP/M	3	<i>INKEY\$</i>	28
CS	47	<i>INPUT</i>	20
<i>CREATE</i>	43	<i>INPUT#</i>	38
Cursor	24	<i>INSTR</i>	24
D		Instructions	11
<i>DATA</i>	21	<i>INT</i>	30
Data base	58, 65	Integrated circuits	2
Data file	42	Interpreter	3
Debugging	22	<i>ITEM</i>	58
Default	7	J	
DEFinition	65	JETSAM	42
<i>DELETE</i>	15	K	
<i>DIM</i>	30	Keyed files	42
<i>DIR</i>	7	Keywords	11
Direct mode	15, 49	<i>KILL</i>	19
<i>DISCKIT</i>	7	Kilobyte	1
E		L	
<i>ED</i>	53	<i>LABEL</i>	59
<i>EDIT</i>	15	<i>LAST</i>	58
<i>ELSE</i>	29	Least significant bit	2
<i>END</i>	29, 53		
<i>ER</i>	53		
<i>ERA</i>	37		

	Page		Page
<i>LEFT\$</i>	23	Recursion	
<i>LEN</i>	23	<i>REM</i>	
<i>LET</i>	19	Remainder	
Lines	52	<i>REMPROP</i>	
LISP	3	<i>RENAME</i>	
List	50, 57	<i>RENUM</i>	
<i>LIST</i>	15	<i>REPEAT</i>	
<i>LLIST</i>	18	<i>RESTORE</i>	
<i>LOAD</i>	11	Result	
Loading	4	<i>RETURN</i>	
Local variables	56	<i>RIGHT\$</i>	
Loop	27, 59	<i>RL</i>	
<i>LOWERS\$</i>	24	<i>RND</i>	
<i>LSET</i>	41	<i>ROUND</i>	
<i>LT</i>	48	<i>RSET</i>	
		<i>RT</i>	
M		<i>RUN</i>	
<i>MAKE</i>	56		
Memory	2	S	
Microprocessor	2	<i>SAVE</i>	11
<i>MID\$</i>	23	Saving	4
Most significant bit	2	<i>SEEKKEY</i>	45
Multiple recursion	62	Semicolon, use of in BASIC	16
		Sequential access	37
N		<i>SETCURSOR</i>	52
Nesting	28	<i>SETPOS</i>	50
<i>NEW</i>	17	<i>SETSPLIT</i>	48
<i>NEXT</i>	27	<i>SETX</i>	50
<i>NODES</i>	56	<i>SETY</i>	50
Null string	28	<i>SHOW</i>	52
		Software	5
O		<i>SPACE\$</i>	24
<i>OP</i>	57	<i>SPC</i>	16
<i>OPEN</i>	37	Spelling checkers	5
Operating system	3	Split screen	47
Operator	12	<i>SS</i>	48
		<i>ST</i>	49
P		<i>STOP</i>	33, 59
PASCAL	3	Stop	17, 55
Pauses	28, 60	Store	2
<i>PD</i>	50	String	15
<i>PE</i>	50	<i>STR\$</i>	23
Pixels	48	<i>STRING\$</i>	24
<i>PLIST</i>	65	<i>SUBMIT</i>	8
<i>PO</i>	52	Subroutine	29
<i>POALL</i>	54	Sub-string	24
<i>PPROP</i>	65	Subtopic	9
<i>PPS</i>	66	Syntax error	15
<i>PR</i>	9, 52	<i>SYSTEM</i>	8
Primitive	11, 48	System discs	3
<i>PRINT</i>	15		
Procedures	47, 53	T	
PROFILE	8	<i>TAB</i>	16
Program mode	15	<i>TF</i>	50
Prompt	7	<i>THEN</i>	29
Property	65	<i>THROW</i>	63
Property pair	65	<i>TO</i>	53
PTR	18	<i>TOWARDS</i>	50
<i>PUT</i>	41	<i>TROFF</i>	18
		<i>TRON</i>	18
Q		<i>TS</i>	48
Quotation marks, use of in BASIC	16	Turtle graphics	47
<i>QUOTIENT</i>	54	<i>TYPE</i>	7, 52
R		U	
<i>RANDOM</i>	60	Unwinding	62
Random Access	40	<i>UPPER\$</i>	24
Random numbers	12	<i>USING</i>	16
Rank	42		
<i>RC</i>	59	V	
<i>READ</i>	21	<i>VAL</i>	23
Reading	3	Variables	12, 19
Read-only memory (ROM)	3		
Recall	2		
Record	40		

W	Page		Page
<i>WEND</i>	28	Writing	3
<i>WHILE</i>	28	Write-protection	3
<i>WIDTH</i>	16		
Workspace	54	Z	
<i>WRITE</i>	38	Zone	16
<i>WRITE#</i>	38	<i>ZONE</i>	16



BERNARD BABANI BP188

Getting Started with BASIC and LOGO on the Amstrad PCWs

- Your Amstrad PCW is more than just a word processor. It is also an extremely powerful and versatile microcomputer capable of innumerable applications in practically every field of endeavour.
 - This book has been written especially to complement the manufacturer's manuals and to help readers overcome the first hurdles in computing on the PCWs. It covers first steps in both BASIC and LOGO and includes simplified exercises for "hands on" experience.
 - "Getting Started" means just that. The book is for beginners and those with a modicum of experience. It is the companion volume to BP187 "A Practical Reference Guide to Word Processing on the Amstrad PCW8256 and PCW8512".
 - An essential addition to the library of all PCW users.
-

£5.95

